Integration of Verification and Testing into Compilation Systems – Concept and Case Study –

vorgelegt von
Diplom-Informatiker
KLAUS DIDRICH
aus Berlin

Von der Fakultät IV
— Elektrotechnik und Informatik —
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
— Dr.-Ing. —

genehmigte Dissertation

Promotionsausschuß:

Vorsitzender: PROF. DR. ADAM WOLISZ
Berichter: PROF. DR. PETER PEPPER
Berichter: PROF. DR. STEFAN JÄHNICHEN

Tag der wissenschaftlichen Aussprache: 6. November 2001

Berlin 2001

D 83

KLAUS DIDRICH

Die Integration von Verifikation und Test in Übersetzungssysteme – Konzept und Fallstudie

(Integration of Verification and Testing into Compilation Systems – Concept and Case Study)

Zusammenfassung

In dieser Arbeit wird die Architektur für einen Compiler vorgestellt, der die *Korrektheit* der übersetzten Quellen *als Teil des Übersetzungsvorgangs überprüfen* kann. Dabei soll es möglich sein, *verschiedene Methoden*, wie etwa formaler Test und formaler Beweis, einzusetzen, um die Korrektheit nachzuweisen.

Ein vollautomatischer Nachweis ist sehr aufwendig und häufig auch gar nicht möglich. Es ist also nicht praktikabel, aus Spezifikation und Programm die Korrektheit automatisch abzuleiten. Wir erweitern daher die Sprache um Korrektheitsnachweise (*justifications*), die der Benutzer in den Quelltext einfügen muß („*literate justification*"). Je nach gewählter Methode muß der Benutzer den Korrektheitsnachweis mehr oder weniger genau ausführen. Durch die Einführung der Korrektheitsnachweise muß der Übersetzer *Beweise nur noch überprüfen* anstatt sie automatisch abzuleiten.

Die Überprüfung der Korrektheitsnachweise kann in den Übersetzer integriert werden oder an ein externes Werkzeug delegiert werden. Ein externes Werkzeug erlaubt die Einbindung bereits existierender Werkzeuge, aber auch eine Neuentwicklung eigener Werkzeuge ist möglich. Wir zeigen am Beispiel eines taktischen Theorembeweisers, daß eine Eigenentwicklung nicht unbedingt aufwendiger ist als die Anpassung eines vorhandenen Werkzeugs.

Um Tests während der Übersetzung durchführen zu können, muß ein *Interpreter* zur Verfügung stehen. Die Ausführung ungetesteten Codes birgt allerdings auch *Sicherheitsprobleme*. Wir diskutieren verschiedene Möglichkeiten, mit diesem Problem umzugehen.

Der *Korrektheit* einer Übersetzungseinheit entspricht in der Semantik die *Konsistenz einer algebraischen Spezifikation*. Wir betrachten zwei Beweismethoden: zum einen durch Konstruktion eines Modells und zum andern durch Nachweis einer korrektheitserhaltenden Relation. Die Beweisverpflichtungen ergeben sich zunächst aus der Beweismethode, außerdem werden Beweisverpflichtungen eingeführt, um die Korrektheit von zusammengesetzten (modularen) Programmen zuzusichern.

Die in dieser Arbeit beschriebene Architektur ist *prototypisch implementiert* worden. Dazu wurde das OPAL-System um Elemente zur Spezifikation und zur Beschreibung von Korektheitsnachweisen erweitert. In der Arbeit werden einige kurze Beispiele vorgeführt. Der OPAL/J-Prototyp ist seit Version `2.3e` Teil der OPAL-Distribution.

KLAUS DIDRICH

Integration of Verification and Testing into Compilation Systems – Concept and Case Study

Abstract

In this thesis we present a compiler architecture that enables the compiler to check the correctness of the source code *as part of the compilation process.* It allows to perform these correctness checks with *different* methods, in particular formal testing and formal proof.

A fully automated check is very expensive and often impossible. Hence, it is not feasible to check correctness automatically with the help of specification and implementation. We extend the programming language by (correctness) *justifications* that the user must insert into the source code (*"literate justification"*). Depending on the chosen justification method the user must work out the justification in more or less detail. The introduction of justifications changes the compiler's task from deriving a correctness proof by itself to *checking a correctness proof* provided by the user.

The correctness check for justifications can be integrated into the compiler or delegated to an external tool. An external tool allows the integration of existing tools but the development of specialized tools is also possible. The example development of a specialized tactical theorem-prover shows that the development of a specialized tool is not necessarily more expensive than the adaptation of an existing tool.

For test execution during the compilation an *interpreter* must be available. The execution of untested code causes *security risks.* We discuss different possibilities to deal with this problem.

The *correctness* of a compilation unit corresponds to the *consistency of an algebraic specification.* We study two proof methods: either by construction of a model or by establishing a correctness-preserving relation. The proof obligations result from the proof method, in addition proof obligations are introduced to ensure the correctness of modular programs.

The compiler architecture described in this thesis has been *prototypically implemented.* The OPAL system has been extended with language elements to denote specifications and (correctness) justifications. The thesis presents some short examples. The OPAL/J prototype is part of the OPAL distribution since version 2.3e

Acknowledgement

I want to thank my advisor, Peter Pepper, for his support and his valuable advice in the preparation of this thesis.

This thesis was done in the context of the OPAL project. Many thanks to the many members of the OPAL Group: Olaf Brandes, Gottfried Egger, Jürgen Exner, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, Michael Jatzeck, Johannes Labisch, Christian Maeder, Wolfram Schulte, and Mario Südholt who developed the OPAL system to its present state.

I would like to thank in particular Wolfgang Grieskamp and Christian Maeder who would always discuss new ideas for verifying OPAL and answer questions about internal details of the OPAL compiler. Without their support the OPAL/J prototype would not exist.

DaimlerChrysler generously supported me with a stipendium since 1999.

Thanks to my parents for their support and their encouragement throughout the past years.

And finally, I would like to thank my wife, Anette, for her patience and her understanding throughout the time I was writing this thesis.

Contents

List of Programs

List of Figures

Chapter 1

Introduction

In recent years, there has been growing interest in proving programs correct, or at least partially correct. As software is increasingly used in safety-critical environments and the security of software programs becomes more and more important, developers are eager to produce correct programs and obtain a certificate of the correctness and "harmlessness" of their software.

Producing correct software is not just a matter of pride for the engineer. United States and European Union law [EEC85] states that the manufacturer of faulty software is liable for damage caused by death, personal injury or damage to private property [KS91, Gün93]. It is therefore worthwhile putting extra effort into the development of correct programs in order to reduce the risk of later liabilities.

Actually, the demand for correctness checks is not really new. Experience shows that it is easier to have errors detected automatically than to find them by debugging. It is not surprising, then, that programming languages are increasingly including features that enable the compiler to check certain correctness aspects at compile time. Two of the most important features are *strong typing* and *interfaces*.

Both features are taken for granted in modern programming languages and it is easily forgotten that typing and interfaces were not available in the early programming languages. FORTRAN 77 and COBOL have only a predefined set of types, the user having to code other types by using the predefined ones. C does have types, but it allows typing rules to be circumvented. Interfaces are not available in the first definition of PASCAL, which provides textual inclusion as the only means to simulate interfaces.

Typing and the use of interfaces are the most common correctness checks found in programming languages. Other checks are dynamically checked assertions (e. g.

in EIFFEL), the check for malicious operations performed by the JAVA byte-code verifier and the check for correctly declared exceptions in JAVA.

It seems quite natural to extend the responsibility of the compiler by including correctness checks for specifications in the compilation process. This is discussed in Chapter 2.

1.1 Motivation

The motivation for this thesis is a strong dissatisfaction with the gap between the potential (functional) programming languages offer for developing correct software and the capabilities of the available tools. Functional programming languages in particular provide a combination of features that make them a good starting point for the development of correct software.

- The link between functional languages and algebraic specifications is close enough to enable much of the algebraic framework developed for handling such specifications to be reused. This was shown in the KORSO project ([BDDG93] and [PBDD95]). We are thus able to reason about programs and specifications within a single framework.
- As regards the practical aspects, compilers for functional programming languages have long emerged from their academic roots and now produce code that is fast enough to be used in practice.
- Some functional programming languages already provide limited support for algebraic properties. ML uses an ad hoc syntax to mark types that have an equality function associated, HASKELL extending this idea by type classes that allow the user to associate a type with arbitrary functions. OPAL supports the implementation of abstract data types by a non-isomorphic implementation. In the KORSO project, syntactic support for algebraic properties was added to OPAL. A sequent calculus for OPAL [DF94] has been developed that makes formal proofs of the properties of OPAL programs possible.

These advantages influenced the decision to focus our work on functional programming languages. Because some of the work required to develop a prototype has already been done, it is easier (not easy!) to arrive at a usable environment for the development of correct software. In principle, however, the approach presented here does not depend on the paradigm of the underlying programming language.

Syntactic support for algebraic properties is a first step, but having *only* syntactic support may turn out to be harmful. From the user's point of view, algebraic

properties belong to the language, so successful termination of the compiler is easily misinterpreted as an indication of the program's correctness. However, the compiler does not check these properties. Violation of unchecked algebraic properties often leads to especially nasty errors, which are typically detected after hours of debugging. We need both, algebraic properties *and* correctness checks.

On the other hand, if users are aware that algebraic properties are essentially treated as structured comments, they will probably not synchronize the specification and the code. This effect is known from the development of documentation. If another programmer discovers that specification and source code do not match, it will be difficult to decide whether the specification has not been updated or whether the code is wrong (or whether both of them need to be corrected). Again, we see that algebraic properties without correctness checks are dangerous.

The idea of adding support for algebraic properties was further pursued in the development of EXTENDED ML and OPAL 2α, but so far both languages lack tool support. EXTENDED ML is a restricted subset of ML that has been augmented to include algebraic properties and a verification semantics[1]. OPAL and OPAL 2α have no verification semantics, but both offer support for formulating algebraic properties.

Correctness is often viewed on the level of individual functions, but it is also important for programming in the large. Functional programming languages support programming in the large by polymorphism, by parameterized structures and by separation of interface and implementation. Example derivations carried out in the KORSO project show that a richer set of algebraic relations is helpful in developing correct programs. Correctness for programming in the large is discussed in Chapter 4.

1.1.1 Examples

The lack of support for specification and verification is not merely a theoretical deficiency without practical impact on everyday programming. On the contrary, without specification and verification some expensive errors can occur.

We illustrate this situation with some small, but nevertheless typical, examples. The syntax we use is similar to that of OPAL 2α, which allows the formulation of the problem (but does not provide a solution). We have added some syntactic sugar to facilitate understanding.

[1] EXTENDED ML is presented in Section 9.1.2.

1.1.1.1 Sorting a List

The first example (Program 1.1) shows the specification and implementation of a single function using pre- and postconditions[2]. In this case, it is easy to spot the mistake in the implementation.

Program 1.1 Sorting lists

```
FUN sort : seq[nat] → seq[nat]
    SPC  sort(S) = T
    PRE  true
    POST S permutation T AND ascending(T)

DEF sort(◊)    == ◊    -- empty sequence is sorted
DEF sort(a::R) == a::sort(R) [!]
```

This is the classical situation for correctness checks: given an implementation, a precondition and a postcondition, check whether the implementation ensures the postcondition, provided the precondition holds.

1.1.1.2 Instantiation of Sets

The second example involves a parameterized structure with a function parameter that is expected to fulfil some properties. Program 1.2 shows the interface and the implementation of a structure of sets, parameterized with an element sort α and an ordering relation $<$ on that sort.

The function parameter of the `Set` structure is required to be an (irreflexive) total order. The requirement must be expressed as a comment without obligation. The implementation of the `incl` function (not shown here) produces an ordered sequence, the implementation of `in` utilizing this property to enhance efficiency. Problems arise if the instantiation is not correct – see Program 1.3. The example uses best-fit pattern-matching to shorten the definition of the ordering relation[3].

The compiler does not check whether an instantiation fulfils the parameter properties. The error in the instantiation of `Set` with $[\texttt{colour}, <]$ passes unnoticed but leads to some surprises when evaluating the `in` function. The intended order is `red` $<$ `green` $<$ `blue`, so let us assume that the representation of the set $\{\texttt{red}, \texttt{green}, \texttt{blue}\}$ is $\texttt{red}::\texttt{green}::\texttt{blue}::\Diamond$.

[2] This example, like many others in the thesis, contains a mistake. This is intentional; we need erroneous examples to illustrate how certain kinds of errors are detected or ignored. These intentional mistakes are marked with [!].

[3] One might argue that ordering relations should be automatically generated anyway. But this is not always desirable, nor does it affect the underlying problem.

Program 1.2 Sets

```
SIGNATURE Set[α, <]
  SORT α                                  -- parameter sort
  FUN <: α × α → bool                     -- parameter function
  -- requirement : (α, <) is an (irreflexive) total order

  SORT set
  FUN {}: set
  FUN incl: α × set → set
  FUN in: α × set → bool

  LAW ALL x S. x in incl(x,S)
  ...
```

```
IMPLEMENTATION Set[α, <]
                  -- implementation of set by (ordered) sequences
  DATA set == abs(repr: seq[α])

  FUN in: α × set[α, <] → bool
  DEF x in abs(◊)   == false
  DEF x in abs(a::R) == IF x < a THEN false
                        IF a < x THEN x in abs(R)
                                 ELSE true
                        FI
  ...
```

Program 1.3 Instantiation of Sets

```
IMPLEMENTATION Colour

  DATA colour == red green blue

  FUN <: colour × colour → bool
  DEF red < x     == true  !
  DEF green < red == false
  DEF green < x   == true  !
  DEF blue < x    == false

  IMPORT Set[colour, <] COMPLETELY
```

```
red in abs(red::green::blue::◊)
  ⇝ false  -- since red < red
green in abs(red::green::blue::◊)
  ⇝ green in abs(green::blue::◊)  -- since red < green
  ⇝ false  -- since green < green
```

Such an error is particularly nasty for several reasons. First, *the error shows up in the functions from the correctly implemented module* `Set`, and not in the function $<$ on colours, the true cause of the error. Second, the error occurs *only for a few special cases*, which might not even show up in every application. Third, the programmer, who is unfamiliar with the implementation of the module `Set`, will have *a hard time realizing that all the errors* in functions from `Set` have a single cause, which, moreover, *lies in the* `Colour` *module.*

1.1.1.3 Implementation of Deques

The last example uses a feature of OPAL that allows the implementation of a free type to be different from its declaration in the interface. Program 1.4 shows how a data structure of "deques" can be implemented. Deques are semantically equivalent to sequences, but allow (amortized) O(1) access to the first and the last element. The idea is well known (see, e. g., [Gri81]): the sequence is represented as a pair of sequences, the second of which is reversed (compare the definition of `asSeq` in the example).

The function `find?` is available for all collection types in the OPAL standard library. It takes a predicate and an aggregate as arguments and returns one element of the aggregate that fulfils the predicate or `nil` otherwise.

This is also the case for the `find?` function on type `deq`, and yet it is not defined correctly. Consider the following evaluations:

```
find?(λx. true, abs(1::2::◊, ◊)) ⇝ avail(1)
find?(λx. true, abs(◊, 2::1::◊)) ⇝ avail(2)
```

Hence, `find?` returns two different results for representations of the same sequence. *It is not even a function.* A correct implementation would have to convert to type `asSeq` before using the `find?` function on sequences.

Errors like this are difficult to detect because the effect of the error shows up only in special circumstances. After all, the explicitly written part of the specification is fulfilled. Because the function property is basic to functional programming, it is usually considered last as the underlying cause.

Program 1.4 Deques

```
SIGNATURE Deque[α]
  SORT α

  TYPE deq == ◊
              ::(ft : α, rt : deq)
  FUN asSeq : deq → seq

  FUN find? : (α → bool) × deq → option[α]
      SPC  find?(P, d) == r
      PRE  true
      POST (avail?(r) ⟹ cont(r) in d AND P(cont(r))) AND
           (nil?(r) ⟹ ALL x. x in d ⟹ not(P(x)))
  ...
```

```
IMPLEMENTATION Deque[α]
  DATA deq == abstract(left : seq, right : seq)

  DEF asSeq(d) == left(d) +|+ revert(right(d))

  DEF find?(P, d) ==
      IF avail?(find?(P, left(d))) THEN find?(P, left(d))
                                   ELSE find?(P, right(d)) [?] FI
  ...
```

1.1.1.4 Discussion of the Mistakes

It should be noted that the second and third example show errors that are likely to be expensive in software development.

- The involved functions *are likely to pass initial tests and applications.* The error thus only becomes evident at a late stage in the development process.
- *The distance between cause and effect is considerable.* Inferring that the wrong results of evaluating the in function are the result of the incorrect implementation of the order on colours takes some time and requires knowledge of the structures involved.

Correctness properties like the ones presented are part of the language definition. A compiler is not only expected to translate the input into a different output language, it is also regarded as one of the principal tools for deciding whether a piece of source code conforms to the language definition. In the cases presented above, the compiler fails to issue an error message, although the programs are incorrect. Of course, the necessary checks are not fully automatable, some user support being needed. But complete lack of support for these checks is not ac-

ceptable. The treatment of these examples using our approach is presented in Chapter 7.

1.2 Justification of Correctness

The examples show that we expect formal specifications to already exist (and that we do not have to deal with the problem of how to translate the user's requirements into a formal specification). A formal specification is only a necessary condition for *automated* checking of correctness, though.

1.2.1 Kinds of Justification

We use the generic term "justification" for any method designed to increase the user's confidence in the correctness of the program. Instead of confining ourselves to only one method, we want a tool that supports a wide range of justification methods.

In particular, we wish to support *formal testing* and *formal proof*. These methods are advocated by different communities to establish correctness. Both methods have their advantages and disadvantages; we do not want to take sides in that discussion.

Our aim is to enable the user to choose the appropriate justification method in each case, and even change the justification method in the development process – for example, from a simple test to a formal proof. The error in the sort function (Program 1.1) will become evident during testing, whereas the error in the implementation of deques (Program 1.4) is likely to pass most tests, thus requiring a formal proof.

For details, see Chapter 6.

1.2.2 Extent of Justification

Justification is an expensive activity. It is therefore desirable that the user be able to decide to what extent the correctness of the software system should be checked. Some typical choices are:

- *No justification.* For backwards compatibility and for the development of quick-and-dirty prototypes, it must be possible to develop software without justifications.

- Some users might want to *check only* certain *"hot spots"* that are critical to system safety or security.
- Others might want to *check only certain kinds of properties*, for example, definedness properties or instantiation conditions.
- And, of course, it must be possible to *perform a full-fledged formal development.*

1.2.3 Feasible Justification

A big problem here is the fact that algebraic properties are in general undecidable, and even the decidable properties cannot easily be proven automatically. One possibility is to restrict either the programming language or the expressible properties such that the resulting combination is automatically provable. Both options are undesirable: weak properties make the resulting system unsuitable for meaningful correctness checks, and restricting the expressiveness of the language results in correct but uninteresting programs. As the examples in Section 1.1 show, correctness issues already exist in data-type implementation, an area we will certainly need for developing software.

Our approach is not to restrict the language. Instead, we propose *requiring the user to add information* that makes automated checking of the necessary justifications feasible.

1.3 The Opal/J Case Study

Based on the existing OPAL compiler, a prototypical implementation of a correctness-aware compiler has been developed [Did99]. The compiler closely follows the design presented in Chapters 5 and 6. Some compromises were unavoidable because a few of the design decisions made for the standard OPAL compiler are too expensive to change. The development of the OPAL/J prototype was a considerable help in validating the design decisions and it also served for gathering experience in the development of language extensions for justification and an accompanying development environment.

The justification component was implemented as a component of the OASYS tool[4]. OASYS served as a testbed for the approach presented here. It was intended to give a first approximation of the tool we have in mind. For proper software development, further improvements are necessary. The output is sometimes cryptic,

[4]The OASYS tool allows compilation and interpretation of OPAL programs to be interactively controlled. Its modular design allows additional components to be added.

efficiency could be improved and some syntactic peculiarities should be cleaned up. Nevertheless, the prototype is functional (in two senses: it is written in a functional programming language, and it works and is usable), and the examples we present in this thesis are taken from concrete sources. Only the syntax has been slightly changed to improve readability.

Below, we use the term "OPAL/J" for the prototype and for the extension of the OPAL language. The "J" is short for justification.

Figure 1.1 on the facing page shows a screen-shot of the OPAL/J prototype during preparation of the examples presented in Chapter 6. The screen-shot displays four windows, three editor windows and one terminal window. The toolbars of the editor windows have four extra buttons that support the extensions of OPAL/J. We begin our explanation of the contents with the upper left-hand window and proceed clockwise.

The **upper left-hand window** shows the OPAL unit `SortList.sign` that declares a function `sort` on sequences of natural numbers and includes a *specification* for this function. If we remove the syntactic sugar, the specification of the `sort` function reads

$$\text{ALL S.}\ \underbrace{\texttt{true}}_{\text{PRE}} \Longrightarrow \underbrace{\texttt{S permutation sort(S) AND ascending(sort(S))}}_{\text{POST}}$$

The specification can be accessed via the name `Spc[sort]`. Experience has shown that the simultaneous introduction of a definedness property is useful. This definedness property is constructed from the precondition only and reads ALL S. `true` $\Longrightarrow$ DFD `sort(S)`. The definedness property can be accessed via the name `Dfd[sort]`.

The directive `/$ PROOFCHECK $/` in the second line tells the OPAL/J compiler that a full-fledged formal development is desired and that the full set of proof obligations is to be computed. If this directive were missing, OPAL/J would assume that no correctness checks should be performed, thus ensuring backwards compatibility.

The **upper right-hand window** presents the corresponding implementation unit `SortList.impl`. First, the function `sort` is defined (incorrectly). Then, some (pseudo) imports are necessary to make the justification extension known to the compiler. These are followed by four justifications of the formula `Lift[Spc[sort]]` using different methods. The lifted specification is in this case isomorphic to the original specification; see Section 3.3.5 for an explanation of why we cannot normally use the original specification. All justifications are preceded by a *proof declaration* that declares which properties are justified and which properties are assumed to hold during justification.

Figure 1.1: The OPAL/J Prototype Justifying the Unit SortList.impl

The Justification `spc_sort_1` uses *certification* as a justification method. The justification consists of a digital signature. Section 6.2 deals with certification as a justification method.

The Justifications `spc_sort_2a` and `spc_sort_2b` attempt to justify the correctness by means of *formal testing*. The parameters for a formal test are the function to be tested, the test-case selector function and the test data. The test-case selector function `allGuards` is a white-box test-case selector ensuring that all guards of the if-branches of the defining equation are covered by at least one item of test data. Justification `spc_sort_2a` contains only one item of test data, the empty sequence, Justification `spc_sort_2b` also contains an ascending and a descending sequence. The testing component is described in Section 6.3.

Finally, Justification `spc_sort_3` contains a *formal proof*. The proof declaration lists several premises that are used during the proof: the defining equation of sort (`Def[sort]`), free-type properties of sequences of natural numbers (`Freetype[seq[nat]]`), and some definedness properties as well as properties about permutations and ascending sequences of natural numbers. The formal proof is composed of three proof strategies. The formal proof component is presented in Section 6.4.

The **lower right-hand window** is the terminal window. It contains the output of the call to the justification component. The first lines contain messages output during the justification check. The time shown is not processor time but wall-clock time. It thus took 6.89 seconds to perform parsing, context checking and justification checking[5]. The justification check fails with several error messages and one hint. We explain the error messages briefly here; they are explained in more detail in the corresponding sections of Chapter 6.

- The first error message lists a proof obligation for which no proof declaration could be found.
- The second error message states that the digital signature is valid but not from a trusted authority.
- The third error concerns Justification `spc_sort_2a`: one test case (expressed as a Boolean-valued lambda expression) is not covered by the supplied test data.
- The fourth error lists a failed test. The test of Justification `spc_sort_2b` fails for the third piece of input data (since we count from 0, the third piece of input data is labelled "no. 2").

[5]The time was measured on an Intel-compatible processor, running at 600 MHz, equipped with 128MB main memory.

- The last error message tells us that the theorem-prover did not finish with an empty list of subgoals. The message lists the targets of the remaining subgoals.
- The hint draws attention to the fact that the theorem-prover might succeed in another branch of the proof tree.

The **lower left-hand window** shows a file in which additional information is stored to facilitate debugging. We see the tracing output of the proof attempt of `spc_sort_3`. The atomic tactics performed are listed, and finally the resulting proof state is given in full, followed by some statistics on the performance of the theorem-prover.

The other examples (instantiation of sets, implementation of deques) are given in Chapter 7.

1.4 Contents

In the following chapter, we motivate once again our decision to integrate verification and testing into the compilation process, and discuss some conclusions. Chapters 3 and 4 give the semantic foundations for a justification component. Chapter 5 presents the design of a compiler architecture with integrated justification support. Chapter 6 focuses on the architecture of the justification component itself. Chapter 7 shows how the examples given in this introduction are handled by the OPAL/J compiler. Chapter 8 discusses security issues related to our approach. Related work is presented in Chapter 9, and further topics are listed in Chapter 10. The thesis concludes with a final discussion.

Chapter 2

The Compiler as a Correctness Checker

Justification of correctness – by formal proof or formal testing – is mostly performed by a separate tool in the development environment. This has several disadvantages. Tools tend to evolve in different directions; even if the compiler and a justification tool were developed for the same input language, we cannot expect the input language to remain 100% compatible. A separate tool must reanalyze the code, whereas the compiler has already analyzed the input. Using the compiler therefore promises to be more efficient than using a separate tool. Finally, integration of the correctness check encourages a development style that views correctness checks as an integral part of software development, whereas a separate tool is often used for a "post-mortem justification" – if it is used at all.

Compilers have been enhanced with other correctness checks before. Section 2.1 explains why we think the *integration of the justification correctness check into the compilation process* is a logical step in compiler development.

Formal proof and formal testing are tasks that are typically performed semi-automatically using interactive tools. Fully automated support is not feasible, so we have to find a way of integrating these interactive tasks into the non-interactive (batch) process of compiling. Our approach to making justification a feasible task for the compiler is presented in Section 2.2, which introduces the concept of *"literate justification"*.

Integration of the justification into the compilation process creates the technical problem of having to execute tests before the tested function has been compiled. To make this possible, we propose (in Section 2.3) the *use of an interpreter* during the compilation process for the justification.

2.1 Extending the Compiler

A compiler has two main tasks. First, it must translate correct source code into the target language. Second, it must check that the source code adheres to the language definition. If the input does not belong to the input language, the compiler should stop compiling and issue appropriate error messages, which help the user to understand the reason for the error.

In the past, compilers have not always done a good job as regards the second task. Of course, unexpected behaviour sometimes results from misunderstandings. The most (in)famous of these misunderstandings is the confusion of assignment and equality. Some languages (e. g. C) use a single equals sign for the assignment operator, which is easily confused with the equality operator. If, in addition, assignment is treated as a function that returns a value interpretable as a Boolean value, the compiler has no chance of detecting an error in a program fragment like '`if (n = 1) then ... else ... fi`'.

While this is a misunderstanding between the language designer and the programmer, a compiler that fails to check correctness conditions can pose problems that are really hard to detect. The example in Program 2.1 is taken from [Pag88].

Program 2.1 An Illegal Subroutine Call in FORTRAN 77

```
SUBROUTINE SILLY(N, M)          CALL SILLY(10, 7)
N = N + M                       WRITE(*,*) 10
END
```

It is illegal to call a subroutine with a constant actual parameter if the corresponding formal parameter is written to within the subroutine. "The effects of committing such an error are system-dependent. [...] some systems will actually go ahead and over-write the constant with a new value, so that if you use the constant 10 in some subsequent statement in the program you may get a value of 17. Since this can have very puzzling effects and be hard to diagnose, it is important to avoid doing this inadvertently." [Pag88]

Formally, this is a compiler error, but FORTRAN 77 actually lacks the means to denote whether the call to the subroutine is wrong or whether the subroutine should be corrected. It is possible to write a compiler that is able to diagnose this kind of error, but this requires an additional analysis, which was obviously not foreseen by all FORTRAN 77 compiler writers[1].

[1] FORTRAN 90 has been extended by interfaces and the option of declaring formal parameters to be `IN`, `INOUT` or `OUT`. The example can now be formulated such as to allow the compiler to detect the error more easily.

2.1.1 Typing and Interfaces

We have seen in the previous section that FORTRAN 77 contains no elements allowing the compiler to check the correctness of the input. Typing and separately compiled interfaces are two important techniques used to enable the compiler to check for further correctness conditions.

Note that these techniques are not normally available in shell programming languages. Shell programs are often short and have low algorithmic complexity. The additional type checks and the management of interfaces involves too much effort in these cases. If shell programming languages evolve into regular programming languages, these concepts are introduced into the language. PERL is an example of this.

Typing Types are known from mathematics, and it is perhaps not surprising that most programming languages are typed languages. Early languages were restricted to a small predefined set of types like numbers and strings (e. g. BASIC) or integer numbers and real numbers (FORTRAN 77). This forces the programmer to express the problems in terms of the predefined types. But if the programmer is allowed to define new types, a more problem-oriented use of types is possible. Since this makes the type constraints tighter, the compiler can perform more meaningful checks.

The flat model of types can be extended by subtyping, which is provided by some functional languages (OBJ, ISABELLE), or the inheritance provided by object-oriented languages. The most extensive type theory is Martin-Löf's intuitionistic type theory [ML84].

Interfaces Interfaces extend type checking to calls of subroutines in separately compiled program parts. Without interfaces, external functions must be independently declared in the calling program unit. All program units are independently compiled and linked together. The linker does check that a function with the given name exists, because it must adapt these calls to the correct address, but it does not check the (type-)correctness of the call. Thus, the language designer must add a language element that enables the compiler to check the calls to subroutines in other program units. To underline the importance of this kind of check, we once again cite [Pag88]: "Errors, particularly mismatches of data type or array bounds, are especially easy to make but hard to detect. Sometimes the only indication is that the program produces the wrong answer. This shows how important it is to check procedure interfaces." We share this opinion but believe it is the job of the compiler, not of the programmer.

2.1.2 Static Program Analyses

Compilers did (and do) not check for all possible mistakes. Software engineers however, had to cope with these errors, and were thus forced to introduce an additional verification phase: static program analysis. This deals with errors that can be detected from analysis of the source code without executing it. The catalogue of static analysis checks in Figure 2.1 is taken from [Som92] and gives an overview of typical checks of this sort.

- Unreachable code
- Unconditional branches into loops
- Undeclared variables
- Variables used before initialization
- Variables declared and never used
- Variables written twice with no intervening assignment
- Parameter type mismatches
- Parameter number mismatches
- Uncalled functions and procedures
- Non-use of function results
- Possible array bound violations
- Misuse of pointers

Figure 2.1: Static Analysis Checks

There are different ways of dealing with these mistakes.

- Sometimes it suffices to add more analyses to the compiler, e. g. to detect unreachable code.
- Or elements of the language are omitted, e. g. to inhibit branches into loops, the whole concept of branches is eliminated.
- Or new concepts (and keywords) have to be introduced. Consider the declaration of variables. Some languages do not require the explicit declaration of variables. But the danger of typographic errors accidentally introducing new variables more than justifies this facility. The compiler cannot check for this error unless declarations are added to the language.

Initially, separate tools were invented for such analyses, e. g. LINT for C programs or FTNCHK for FORTRAN 77 code. These checks were later added to modern compilers as well. Of course, without changes to the language, only the first two options can be used. In some cases, the language definitions have been extended to allow further checks.

To illustrate this, we look at how parameter type mismatches are handled in C. Program 2.2 gives an example of a legal[2] C program in which a function `strange`

[2]Note that the FORTRAN 77 program in Program 2.1 is *illegal* – though this is difficult to check.

is called with too many – and also wrongly typed – parameters. The language explicitly allows the type conversions implicit in the call. C does not explicitly require that the number of formal and actual parameters match, and it encourages a coding scheme that elegantly handles additional parameters – these are effectively ignored. Hence, the program compiles without error messages and even runs without problems[3].

Program 2.2 A "Correct" Program with Parameter Type Mismatches

```
int main(){
  int r;

  r = strange(134514188, "foo", 8);
  printf("result is %d\n", r);
  return 0;
}

int strange(char *a, int b){
  return strlen(a) + b;
}
```

ANSI C includes (explicit) function declarations (called "function prototypes"). If we include the proper function prototype for `strange`, the compiler detects the errors:

```
warning: passing arg 1 of 'strange' makes pointer from integer without a cast
warning: passing arg 2 of 'strange' makes integer from pointer without a cast
too many arguments to function 'strange'
```

Note that the parameter type mismatches are only warnings, the C language definition stating that integers may be converted into pointers and vice versa. This kind of legacy burden is difficult to avoid[4] because language definitions have to be changed compatibly so as not to break existing code. Newly defined languages can start from scratch and avoid these pitfalls.

2.1.3 Correctness Context Conditions

The development, as presented in the previous sections, shows that it is quite natural to demand increasingly sophisticated correctness checks of the compiler. The designers of EUCLID put it this way: "We see it as a (perhaps eccentric)

[3] It may be necessary to adjust the first argument to the call of `strange` to avoid run-time errors. The result is the meaningless value `134514173`.

[4] C++ still has the feature of undeclared functions, though marked as "anachronistic" [Str87].

step along one of the main lines of current programming language development: transferring more and more of the work of producing a correct program, and verifying that it is consistent with its specification, from the programmer and the verifier (human or mechanical) to the language and its compiler." [PHL+77]

Program 2.3 shows that we still have a long way to go. It contains a recent example: the `Comparable` interface of the JAVA 2 SDK. The documentation contains several requirements for the implementation that "the implementor must ensure". JAVA cannot express requirements formally, therefore the only way to integrate requirements is a comment. Of course, the JAVA compiler does not check comments for correctness.

Program 2.3 The `Comparable` Interface of the JAVA 2 SDK

```
public interface Comparable {
  public int compareTo(Object o)
  /** Compares this object with the specified object for order. Returns a
  negative integer, zero, or a positive integer as this object is less than, equal
  to, or greater than the specified object.
  The implementor must ensure sgn(x.compareTo(y)) ==
  -sgn(y.compareTo(x)) for all x and y. (This implies that x.compareTo(y)
  must throw an exception iff y.compareTo(x) throws an exception.)
  The implementor must also ensure that the relation is transitive:
  (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.
  Finally, the implementer must ensure that x.compareTo(y)==0 implies that
  sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.
  It is strongly recommended, but not strictly required that
  (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class
  that implements the Comparable interface and violates this condition should
  clearly indicate this fact. The recommended language is "Note: this class has
  a natural ordering that is inconsistent with equals." */
}
```

While the checks from Figure 2.1 have been integrated into modern languages and compilers, we are still left with the need to check for

- correct function implementations
- correct instantiations
- correct data-type implementations

Consequently, we add a ***new set of static analysis checks***.

More analyses are not enough, because the compiler cannot guess the intended specification. We cannot restrict the language either, because we definitely want

to check the correctness of function implementations and data-type implementations. We must therefore add new language elements to handle these static analyses.

Currently, correctness requirements cannot be expressed in most languages. Some offer support in writing specifications but restrict properties to Boolean expressions (EIFFEL, EUCLID). Algebraic properties (such as neutral elements, associativity, etc.) and relations between (parts of) modules are not normally considered in programming languages.

The first task, then, is to define a language that allows the formulation of properties. This is not the hardest part. Algebraic specification languages can serve as a model for formulating algebraic properties. Languages like EXTENDED ML and OPAL 2α show that the syntactic inclusion of algebraic properties is unproblematic.

However, the addition of algebraic properties to a programming language poses a new series of design questions. The approach chosen in OPAL/J is a minimal approach: we added only the necessary properties to the existing syntactic entities. This gives a clear concept but requires explicit coding of many almost self-evident properties. Section 5.2 presents the OPAL/J syntax. Section 10.1 discusses language design issues.

The second task is to implement decision procedures to determine the correctness of a program. This is quite difficult: a major difference from previously employed context checks is that correctness is generally undecidable. This problem is tackled in the following section.

2.2 "Literate" Justification

Given the undecidability of correctness checks, an automatic correctness check cannot always terminate *and* give a correct answer. A non-terminating correctness check is unacceptable for integration into a compiler. We must therefore lower our requirements, and opt for less automation or less correctness (or both).

Less correctness sounds impossible, but it is in fact a viable alternative. Testing is a "less correct" correctness check: if testing fails, we know that there is an error; if testing succeeds, the program might or might not be correct.

Less automation calls for more help from the programmer, who must supply additional information that the compiler can use to check the correctness.

We propose ***adding the justification to the source code***. This means that all information related to a single function is contained in the same piece of

source code. This principle is borrowed from the literate programming approach for documenting software. It ensures that our approach remains feasible because we do not require that the justification be computed by a program. All we require is that *the justification be falsifiable by the compiler.*

- For justification by testing, the test data must be provided by the programmer. More sophisticated test methods may also require a description of the test cases to make it possible to check whether each of the test cases is covered.
- A simple proof checker requires the user to provide a full description of a proof, which is then checked for syntactic correctness and whether it constitutes a valid proof.
- More strongly automated proof tools in the style of, say, ISABELLE are controlled by tactics and possibly user-defined strategies. The user need not give the full proof but instead programs a tactical theorem-prover.

2.2.1 Literate Programming

Literate programming was introduced by Knuth [Knu83] to produce better documented software, software that is "fun to read". The notion of "literate" programming was chosen to make other programs seem "illiterate" by comparison, much as structured programming is superior to unstructured programming.

The most distinctive feature of Knuth's WEB system is its mixture of documentation and program code within the same document. This encourages programmers to change the documentation as they change the program, thus always keeping it up to date. The production of documentation and the compilation of the program are performed by two filters, WEAVE and TANGLE. The WEAVE program generates the pretty-printed documentation. The TANGLE filter strips away the documentation parts, re-orders the program code as required, and then calls the compiler proper.

The Dosfop system The literate programming principle has enjoyed only limited success, but my experience with the development of the DOSFOP documentation tool (together with Torsten Klein, see [DK96]) shows that it is possible to deal successfully with the acceptance problem. Knuth's original WEB system suffers from two deficiencies:

- Using the WEB system requires more conviction than merely adding documentation to source code in a programming language. The documentation system not only adds a typesetting language for the documentation part, it

also changes the underlying programming language. While this might be an improvement, it effectively requires the user to adapt to a new programming language.

- The WEB system does not support modular systems or hierarchically organized modules. This makes WEB-type documentation systems unsuitable for complex software systems.

The DOSFOP system tackles the acceptance problem by observing the following principles:

- Upwards compatibility: old and undocumented code is easily integrated.
- Multiple presentations: the documentation is available in printed and in hypertext form.
- Flexibility: the documentation system is highly configurable.

The last point would seem to be of particular importance. In the last article of a series on literate programming, C. van Wyk writes that a "fair conclusion from my mail would be that one must write one's own system before one can write a literate program" [VW90]. We conclude that most literate programming systems are too specialized for general use.

2.2.2 Consequences for the Justification Component

These principles influenced the proposed design of the justification component. Upwards compatibility is ensured by the demand that quick-and-dirty programs still be allowed. Instead of multiple presentations, we require support for multiple justification methods. Flexibility is ensured by different choices as to what extent correctness should be checked.

The following principles are adaptations of design principles of DOSFOP that proved useful in the development of the documentation system and that we intend to apply to the integration of the justification component.

- The introduction of justifications must *not change the division into compilation units.* In order to adhere to the principle of literate programming, the hints to the testing or the formal proof component must be *recorded in the program source.*
- *Self-evident facts must not require extra work.* "Extra work" means *any* change to the source code. Experience with literate programming systems shows that a new tool must recognize facts that are considered self-evident if it is to gain acceptance.

- The extension for handling correctness proofs must *not influence the specification or programming environment.* In particular, it should be possible to prove only parts of the program (for example, those modules that contain safety-critical functions).

2.3 Integrating an Interpreter

One of the distinctive features of our approach is the ***use of an interpreter during the justification check***. This is only sketched here; we discuss the arguments for and benefits of this idea in the following sections:

- An interpreter for test execution (Section 6.3)
- An interpreter as a universal external tool (Section 6.4)

2.3.1 An Interpreter for Test Execution

The implementation of support for formal testing poses a major problem: How can the compiler execute the function to be tested before the code for this function has been generated? If testing is performed after compilation, the object code is available and can be executed, but this is not really an integration into the compilation process.

The problem can be solved if we base test execution (and evaluation) on the (annotated) abstract syntax. This is possible if an *interpreter* is available: We can execute (or rather interpret) the definition and then feed the result, together with the specification, into a test evaluator, all before the backend of the compiler is called.

To be really useful, the interpreter must have certain properties: it must be fast, provide a faithful translation, and be secure. This is discussed in Section 6.3.5, where we also present the architecture of the testing component.

2.3.2 An Interpreter as a Universal External Tool

The principle of literate justification requires that the user be able to add the justification to the source program, close to the property that is to be justified. Extending the compiler by support for various justification methods often suggests the use of (existing) third-party tools, most notably theorem-provers, but also tools for the computation of test cases. These third-party tools are controlled by an input language of their own, which the user must learn in addition to the

programming language and which cannot be context-checked by the compiler of the programming language.

We suggest providing the *implementation of the external tool* in the form of a library *written in the same language* that we are compiling. This has several advantages:

- The user can develop the software project and control the external tool using the same language.
- In particular, users can *extend* the library by adding their own functions. If a new test-case selection strategy is needed or a new tactical for a formal proof is required, users can simply add the appropriate definition to the program.
- We get the syntax and context check of the justification, i. e. of the control language for the external tool, for free.

Functional programming languages are particularly well suited to define embedded domain-specific languages [Hud98]. In fact, ML was developed from the control language of the LCF system [Pau87, Section 1.1.1].

The disadvantage is that we have to provide the implementation and cannot use existing third-party tools. On the other hand, a specialized tool can better support the specific programming language. The translation into and from the input language of a third-party tool does not come for free either.

We discuss these issues in detail in Section 6.4.4, where we compare the benefits of using a third-party theorem-prover and a specialized theorem-prover for supporting formal proofs as a justification method.

Chapter 3

Semantic Foundations

As semantic foundation we choose algebraic specifications. Algebraic specifications have several advantages that make them a good choice for our purpose. In particular, functional programs *are* algebraic specifications with a restricted notion of formulas. Another advantage is that correctness issues have been studied in connection with algebraic specifications and also with respect to the composition of algebraic specifications. By the use of algebraic specifications as a semantic basis we inherit many of the correctness results.

The unreflected transfer of ideas from the area of algebraic specification sometimes leads to expensive context conditions. Since we are concerned with *feasible* procedures, we have to be careful. We are more willing to compromise and sacrifice some expressive power, if necessary.

3.1 Basic Notions from Algebraic Specification

A full introduction to the field of algebraic semantics can be found in [EM85, EM90]. The following definitions leave out some details that are dependent on the underlying institution, most notably the exact definition of operations (and functionalities), of formulas, and of the satisfaction relation "$\models$".

Simple programs are quite easily translated to algebraic specifications. But some features commonly found in functional programming languages, like higher-order functions, partial functions, and products as result types are often not dealt with in papers about algebraic specification.

[Nic95] presents an approach for these problems. Unfortunately the proper treatment of partiality and higher-order functions complicates the definitions of basic notions like functionality, homomorphism and subalgebra. To shorten the pre-

sentation and to ease understanding, we stick to the standard case and refer to [Nic95] for the extended definitions. This is only acceptable because our results do not depend on the exact definitions. If we were to define the algebraic semantics of a programming language, the situation would be different.

3.1.1 Signatures

An (algebraic) *signature* $\Sigma = (S, OP)$ consists of a set of sort names S and a set of operations OP. Each operation op has an associated functionality fc_{op}. A *signature morphism* $m : \Sigma_1 \to \Sigma_2 = (m_S, m_{OP})$ is a pair of mappings, such that $m_S : S_1 \to S_2$ maps sorts and $m_{OP} : OP_1 \to OP_2$ maps operation symbols such that the continuation m^* of m on the functionalities is respected, i. e. $fc_{m(op)} = m^*(fc_{op})$. A Σ-*algebra* $A = ((A_s)_{s \in S}, (A_{op})_{op \in OP})$ is a family of carrier sets, one for each member of S together with matching operations. Let Σ_1, Σ_2 be two signatures with $\Sigma_1 \subseteq \Sigma_2$, and let A be a Σ_2-Algebra. We define the Σ_1-*reduct* of A wrt. Σ_1, written $A_{/\Sigma_1}$, by $A_{/\Sigma_1} = ((A_s)_{s \in S_1}, (A_{op})_{op \in OP_1})$, i. e. we copy those carrier sets and operations that correspond to items from the smaller signature. Let A and B be two Σ-Algebras. A Σ-*homomorphism* $h : A \to B$ is a family of functions $(h_s)_{s \in S}, h_s : A_s \to B_s$, such that for all operation symbols op we have $h \circ op_A = op_B \circ h$. B is called a *subalgebra* of A, written $B \subseteq A$, if $B_s \subseteq A_s$, for all $s \in S$ and $B_{op} = A_{op}$ for all $op \in OP$.

3.1.2 Specifications

A *(algebraic) specification* $\mathcal{S} = (\Sigma, F) = (S, OP, F)$ consists of a signature together with a set of formulas F. An $\mathcal{S}$-*algebra* is a Σ-Algebra A that satisfies the formulas, i. e. $\forall f \in F \bullet A \models f$ or shorter, $A \models F$. A Σ-homomorphism between two $\mathcal{S}$-algebras is called an $\mathcal{S}$-*homomorphism*. The *reduct* construction is also possible for $\mathcal{S}$-algebras.

Let F_1, F_2 be two formula sets over the same signature. We write $F_1 \overset{F}{\Longrightarrow} F_2$, iff for all $A \in Alg_\Sigma$ we have $(A \models F_1) \implies (A \models F_2)$. A *specification morphism* $m : \mathcal{S}_1 \to \mathcal{S}_2$ is a signature morphism $m : \Sigma_1 \to \Sigma_2$ that respects the formulas, i. e. $F_2 \overset{F}{\Longrightarrow} m^*(F_1)$ (where m^* is the continuation of m on formulas).

These definitions do not make any assumptions about the formula language, in particular they do not assume that an implication between two formula sets exists. This is the reason for the introduction of the "$\overset{F}{\Longrightarrow}$"-relation. However, the formulas we normally employ do have an implication. Therefore we drop the "F" superscript in the following.

3.1.3 Categories

The class of all Σ-algebras is denoted Alg_Σ. The category of Σ-algebras and Σ-homomorphisms is called Cat_Σ. The class of all $\mathcal{S}$-algebras is $Alg_\mathcal{S}$. The category of $\mathcal{S}$-algebras and $\mathcal{S}$-homomorphisms is denoted $Cat_\mathcal{S}$. The category of specifications and specification morphisms is called $SPEC$.

If the members of $Alg_\mathcal{S}$ are identical up to isomorphism, we call the specification *monomorphic*, otherwise the specification is *loose*. If $Alg_\mathcal{S}$ is the empty set, the specification is *inconsistent*. An algebra $A \in Alg_\mathcal{S}$ is also called a *model* of $\mathcal{S}$.

3.1.4 Functors

There are several functors that play a special role in the definition of the semantics of functional programs:

- Let $\mathcal{S}$ and $\mathcal{S}'$ be two specifications with $\mathcal{S} \subseteq \mathcal{S}'$, let $i : \mathcal{S} \to \mathcal{S}'$ be the corresponding inclusion morphism. The *forgetful functor* $\mathcal{V}_i : Cat(\mathcal{S}') \to Cat(\mathcal{S})$ maps all algebras to the corresponding reduct algebra.
- Let $\mathcal{S}$ and $\mathcal{S}'$ be two specifications with $\mathcal{S} \subseteq \mathcal{S}'$, let $i : \mathcal{S} \to \mathcal{S}'$ be the corresponding inclusion morphism. The *restriction functor* $\mathcal{R}_i : Cat(\mathcal{S}') \to Cat(\mathcal{S}')$ maps each algebra A to the smallest subalgebra $B \subseteq A$ such that $\mathcal{V}_i(A) = \mathcal{V}_i(B)$. (The smallest subalgebra always exists and is uniquely determined.)
- Let $\mathcal{S}$ be a specification, $\approx$ be the F-induced equivalence relation on $\Sigma(X)$-terms, such that $(t \approx t')$ **iff** $(F \implies t \equiv t')$ for all $t, t' \in T_\Sigma(X)$. The *identify functor* $\mathcal{I}_\approx : Cat(\mathcal{S}) \to Cat(\mathcal{S})$ maps each algebra A to the quotient algebra $A_{/\approx}$.

Note that the construction of the forgetful functor and the restriction functor is possible for arbitrary specification morphisms. An informal explanation of these functors is given in Section 3.3.4.

3.2 Application to Functional Programs

We already mentioned that it is not the aim of this thesis to define the algebraic semantics of a functional programming language. As far as OPAL is concerned, we refer the reader to [DFG+94], and also to [WDC+95], where an institution for OPAL is presented.

3.2.1 Units

In order to abstract from concrete programming languages, we do not employ notions like modules, structures and the like. We follow the use in the KORSO project and speak of (programming) *units*. A programming unit U normally is also a compilation unit.

We assume that the ***semantics of a unit U is a specification*** $\mathcal{S}_U = (S, OP, F)$. Note that the specification is not necessarily monomorphic. Note also that functional programs are included as a borderline case.

3.2.2 Correctness is Consistency

We have to find an interpretation for "correctness" of a single specification, because correctness normally is defined with respect to another entity.

Informally speaking, we consider a unit U (with semantics $\mathcal{S}_U$) incorrect, if there is a contradiction between two formulas, for example, between a definitional equation and an algebraic property. Such a contradiction leads to an empty set $Alg_{\mathcal{S}_U}$, because no algebra can satisfy contradictory formulas.

This consideration motivates our definition: ***A programming unit is correct, iff its semantics is a consistent algebraic specification.***

3.2.3 Classification of Formulas

For algebraic specifications, there is no need to distinguish between different kinds of formulas. For correctness proofs, i. e. consistency proofs, we need to classify formulas. We employ two independent classifications, one of them syntactic, the other one semantic. The idea is that we need the semantic classification to ease consistency proofs, and that we can derive the semantic classification from a syntactic one.

We only define the different classifications here and study the relationship in Section 3.4.1, because we have to discuss the methods to prove consistency first (in Section 3.3).

On the one hand, we make a *semantic* distinction between essential and insignificant formulas, i. e. between ***axioms*** and ***theorems***. The formula set is partitioned into two subsets, the axioms F_{AX} and the theorems F_{TH}, such that the theorems follow from the axioms: $F_{AX} \implies F_{TH}$. Consistency proofs can concentrate on the axioms set: Let $\mathcal{S}_U = (S, OP, F_{AX} \cup F_{TH})$ be the semantics

of the unit, let $\mathcal{S}'_U = (S, OP, F_{AX})$ the semantics without the theorems, then $Alg_{\mathcal{S}_U} = Alg_{\mathcal{S}'_U}$.

On the other hand, we classify formulas *syntactically* by their role in (functional) programming:

Constructive Formulas The constructive formulas are those that are used by the compiler for the generation of the executable program. Typically, constructive formulas are definitional equations.

Algebraic Formulas Algebraic formulas are all other formulas explicitly given by the user in the source code of the unit.

External Formulas External formulas are introduced from other programming units. Typically, external formulas are introduced by an import. External formulas (F_E) are introduced in Chapter 4.

Relational Formulas If the source code contains the claim that the unit `U` is related to another unit `U'` by an algebraic relation, relational formulas are added to `U`. See Section 3.3.2 for an explanation.

We denote these four disjoint subsets of F with F_C, F_A, F_E and F_R respectively.

3.2.4 Interface and Implementation

OPAL and other programming languages separate the interface and the implementation without regarding the two as belonging to different name spaces. For reasons of simplicity, we do not follow OPAL in this respect.

We regard the units SIGNATURE `Colour` and IMPLEMENTATION `Colour` as two distinct units with two different specifications (S^S, OP^S, F^S) and (S^I, OP^I, F^I) respectively. In particular, the algebraic signatures are not related by a subset relation, i. e. $(S^S, OP^S) \not\subseteq (S^I, OP^I)$.

Of course, there is a close relationship between both signatures, which is expressed by the existence of an injective morphism $\mathbf{i}$ that maps sorts and operations from the signature to their counterparts with equal names (but non-equal *origins*) in the implementation. We will refer to this morphism as a *pseudo inclusion*, because – technically speaking – it is not a real inclusion. So we have $\mathbf{i}(S^S, OP^S) \subseteq (S^I, OP^I)$. Note that this does not carry over to the formulas: neither $\mathbf{i}(F^S) \subseteq F^I$ nor $F^I \Longrightarrow \mathbf{i}(F^S)$ hold in general.

Both units are related by an algebraic implementation relation. We will treat this relation in Section 3.3.4.

3.2.5 An Example Program

The example in Program 3.1 introduces an abstract data type `colour`. Note that Program 3.1 contains two units, the interface unit SIGNATURE `Colour` and the implementation unit IMPLEMENTATION `Colour`[1].

Program 3.1 The Data Type `colour`

```
SIGNATURE Colour
  TYPE colour == red green blue

IMPLEMENTATION Colour
  IMPORT Real ONLY real 0 1 − =

  DATA colour == rgb(r: real, g: real, b: real)

  DEF red   == rgb(1, 0, 0)
  DEF green == rgb(0, 1, 0)
  DEF blue  == rgb(0, 0, 1)
  ...
```

The semantics of SIGNATURE `Colour` consists of a single sort `colour`; six operations, the constructors {`red`, `green`, `blue`} and the discriminators {`red?`, `green?`, `blue?`}; and the algebraic formula `Freetype[colour]`.

The semantics of IMPLEMENTATION `Colour` consists of the sorts {`real`, `colour`}, the operations {`0`, `1`, `−`, `=`, `rgb`, `r`, `g`, `b`, `rgb?`, `red`, `green`, `blue`, `red?`, `green?`, `blue?`}, the constructive formulas {`Datatype[colour]`, `Def[red]`, `Def[green]`, `Def[blue]`}, some external formulas introduced by the import (e. g. NOT $0 \equiv 1$), and the relational formula `Lift[Freetype[colour]]`, which is introduced because Unit `Colour.impl` is related by an implementation relation to `Colour.sign`.

3.3 Proving Correctness

A proof of consistency is not an easy task. The following two possibilities to prove consistency of a specification $\mathcal{S}$ are feasible in our setting:

- We can prove the consistency by constructing a *model* of $\mathcal{S}$ as a witness, i. e. we prove that $Alg_{\mathcal{S}}$ is non-empty by constructing an algebra A and proving that $A \models F$.
- Another possibility is to establish a *correctness-preserving relation* $\mathbb{R}$ between $\mathcal{S}$ and another unit with specification $\mathcal{S}'$. A relation is correctness-preserving, if $\forall\ \mathfrak{s}\ \mathfrak{s}' \bullet ((\mathfrak{s}\ \mathbb{R}\ \mathfrak{s}') \wedge \text{consistent}(\mathfrak{s}')) \implies \text{consistent}(\mathfrak{s})$ holds. If

[1] We will treat this example in more detail in Section 7.3.

we know that $\mathcal{S}'$ is consistent and if we have established $\mathcal{S}\,\mathbb{R}\,\mathcal{S}'$, we know that $\mathcal{S}$ must be consistent as well.

3.3.1 Proving Correctness by Constructing a Model

Constructing a model seems quite difficult, but in our environment this is actually easy. For the constructive formulas, an automatic consistency check is possible and is in fact carried out by the compiler: if they are inconsistent, the compiler terminates with error messages, otherwise the compiler terminates successfully. If we can also prove that the algebraic formulas follow from the constructive ones, we have proven consistency.

Put into mathematical formulas, let $\mathcal{S} = (S, OP, F_C \cup F_A)$ be the specification, F_C being the constructive formulas and F_A being the algebraic formulas. If the compiler terminates successfully, we know that $\mathcal{S}' = (S, OP, F_C)$ is consistent. If we have additionally $F_C \implies F_A$, then we have $Alg_{\mathcal{S}} = Alg_{\mathcal{S}'}$.

We loose some expressive power, because the only specifications for which we can use this kind of consistency proof are those specifications that are fully determined by their constructive formulas. Hence, the algebraic formulas exhibit properties that follow from the constructive formulas, but they do not add new properties. Since we started with a functional programming language and added the possibility to add algebraic properties, this is no real loss. Actually, this restriction is partly overcome by the possibility to synthesize programs from proofs of their respective specifications (see Section 6.5).

We summarize: proving correctness requires to prove that $F_C \implies F_A$, or put in other words, ***proving correctness by constructing a model requires to prove that the algebraic formulas are theorems.***

3.3.2 Proving Correctness by an Algebraic Relation

Proving correctness by an algebraic relation $\mathbb{R}$ allows to replace the consistency proof by the proof that an algebraic relation holds. Together with the knowledge that another specification $\mathcal{S}'$ is consistent and the meta-knowledge that the relation $\mathbb{R}$ holds only between consistent units, we can conclude that $\mathcal{S}$ is correct, if $\mathcal{S}\,\mathbb{R}\,\mathcal{S}'$ holds. The proof that establishes the algebraic relation might be more constructive and thus better automatable than the consistency proof.

The claim that both units are related is a separate (meta-) property of the two units. Hence, the most direct transfer into concrete syntax is the introduction of a new type of unit that serves to declare an algebraic relationship between the "basic" units. The KORSO development graph [PW95] makes a distinction

between units (nodes) and relations (edges), and both may be given justifications. This distinction is clean but requires the introduction of a new type of (meta-)unit into the programming language. The principles stated in Section 2.2.2 demand that no new units are introduced, so we try to prove the relation without the introduction of new units.

Let $\mathcal{S} = (S, OP, F)$. If we want to prove that $\mathcal{S}\,\mathbb{R}\,\mathcal{S}'$ holds, we try to find a set of formulas $F_{\mathcal{S}\mathbb{R}\mathcal{S}'}$ such that $(\mathcal{S}\mathbb{R}\mathcal{S}')\textbf{iff}(F \implies F_{\mathcal{S}\mathbb{R}\mathcal{S}'})$. This is equivalent to the proof that the formulas $F_{\mathcal{S}\mathbb{R}\mathcal{S}'}$ are theorems. This motivates our proposal: we propose to ***prove correctness by algebraic relation by adding relational formulas as theorems to one of the related units***. Preferably the relational formulas are added to the unit that is developed later.

The relational formulas depend on the kind of algebraic relation. We will study three algebraic relations.

- A very simple relation is the equivalence relation (Section 3.3.3).
- The most important relation is the *(algebraic) implementation*[2]. We study this relation thoroughly in Sections 3.3.4, 3.3.5, and 3.3.6.
- Finally, we discuss the interpretation relation (in Section 3.3.7).

3.3.3 The Equivalence Relation

Let `U` be a unit, the correctness of which is to be proven by establishing an equivalence relation to a unit `U'`. Since both units have different signatures, we cannot directly prove $F \implies F'$ and $F' \implies F$. The user must provide a *bijective morphism* $m_{\leftrightarrow}$ that establishes the correspondence between the two signatures.

Let $F = \{f_1, f_2, \ldots, f_k\}$ and $F' = \{f'_1, f'_2, \ldots, f'_l\}$ be the respective formula sets, then the relational formulas are $m^*_{\leftrightarrow}(\bigwedge F) \implies f'_j$ for $j = 1, 2, \ldots, l$ and $\bigwedge F' \implies m^*_{\leftrightarrow}(f_i)$ for $i = 1, 2, \ldots, k$, where $m^*_{\leftrightarrow}$ is the continuation of the bijection $m_{\leftrightarrow}$ on formulas (see Section 3.1.2).

3.3.4 The Implementation Relation

Algebraic implementation is the semantic relation that corresponds to the development by stepwise refinement. Hence, the complexity of the implementation relation depends on the complexity of the refinement notion in the development

[2] Algebraic implementation is not to be confused with the implementation of an algorithm. The former is a relation between two units and neither of them needs to be executable. The latter refers to expressing the algorithm in an executable programming language.

environment. The general idea is that the implementation restricts the model class of the base specification, without yielding an inconsistent specification.

Let $\mathcal{S} = (S, OP, F)$ and $\mathcal{S}' = (S', OP', F')$ be two specifications (the semantics of two units, U – the interface unit – and U′ – the implementation unit), $\mathcal{F} : Cat_{\mathcal{S}'} \to Cat_{\mathcal{S}}$ be the functor that expresses the algebraic relationship between the interface and the implementation, then we define

$$\mathcal{S} \text{ is implemented by } \mathcal{S}' \textbf{ iff } (\mathcal{F}(F') \implies F) \wedge (Alg_{\mathcal{S}'} \neq \emptyset)$$

In order to prove the implementation of $\mathcal{S}$ by $\mathcal{S}'$, we have to show that the implementing unit is consistent and that the formulas of the implementation – translated by $\mathcal{F}$ into the context (i. e. specification) of the interface – imply the formulas of the interface. In the simple case, where stepwise refinement consists of adding new formulas, and the signatures of $\mathcal{S}$ and $\mathcal{S}'$ are equal, no translation is necessary and $\mathcal{F}$ is the identity functor. However, stepwise refinement usually allows other programming techniques as well:

- The implementation U′ may use auxiliary sorts and functions that are not visible (hidden) in the interface U.
- The implementation of a data type t by a data type t′ may contain junk, i. e. U′ may contain elements that are not the representation of any element of t.
- The implementation of data type t by data type t′ may be done in a way that one element of t has multiple representations in data type t′.

Each of these techniques is matched by a corresponding functor in the definition of $\mathcal{F}$ (where $\mathbf{i}$ is the pseudo inclusion from $\mathcal{S}$ to $\mathcal{S}'$):

- Hidden Functions: The forgetful functor $\mathcal{V}_{\mathbf{i}}$ removes these additional items from the semantics.
- Unreachable elements: The restriction functor $\mathcal{R}_{\mathbf{i}}$ removes unreachable data elements.
- Multiple representations: The identify functor $\mathcal{I}_{\approx}$ maps multiple representations to a single element.

OPAL uses the composition of these functors to express the implementation relation [DFG+94]: $\mathcal{F}_{\text{OPAL}} = \mathcal{V}_{\mathbf{i}} \circ \mathcal{R}_{\mathbf{i}} \circ \mathcal{I}_{\approx}$.

3.3.5 Re-Definition of the Implementation Relation

In the form presented in the previous section, a proof for correct implementation is not possible in our approach. The proof as sketched above requires the proof of

$\mathcal{F}(F') \implies F$. These formulas are built over the signature of the implemented unit. This has several disadvantages. First, this results in a correctness proof that uses information from the implementation. This violates software engineering principles. Even worse, it is impossible to justify these proof obligations by testing. The reason is that testing requires an executable that does not exist for U. Finally, the interface most often is developed before the implementation exists. We should add the relational formulas to the newly developed unit, and not be forced to change an already existing unit.

It seems less problematic to ***translate the formulas of the interface instead***. We call this translation function $\mathcal{F}^{-1}$. The implementation relation then reads

$$\mathcal{S} \text{ is implemented by } \mathcal{S}' \text{ iff } (F' \implies \mathcal{F}^{-1}(F)) \wedge (Alg_{\mathcal{S}'} \neq \emptyset)$$

This translation is still not easy to define – it essentially introduces explicit definitions of the domains of the variables in the formulas of the interface in terms of the implementation. The translation is treated in [BH96, BH98] where it is called "*lifting*", and must of course be adapted to the logic used.

There still is the problem that formulas have to be lifted at all. The user must be aware that the lifted formulas may syntactically differ from the representation in the interface. Fortunately there are some (not so rare) cases where we can offer the user to employ the syntactic copy of the proof obligation, because in these special cases the following implication holds: $\mathbf{i}(f) \implies \mathcal{F}^{-1}(f)$. If no new sorts are defined, $\mathcal{F}$ is essentially the forgetful functor and some other simplifications apply.

In Section 7.3 we present an example that illustrates the difference between the lifted formulas and their syntactic copies.

3.3.6 Data-Type Implementation

A data-type implementation may introduce junk elements and multiple representations. This opens the possibility that an image $(\mathcal{F}(A))_{op}$ of an implemented function A_{op} is not a function. In order to exclude this error we also lift the function property for the implemented functions to the implementing specification.

We do this separately for junk elements by adding closedness formulas and for multiple representations by adding congruence formulas. Note that these formulas are only needed for operations contained in the interface. These formulas are not needed for hidden functions.

The examples in Sections 7.2 and 7.3 illustrate the additional closedness and congruence proof obligations.

Junk Elements The implementation of functions from the interface must be closed on the non-junk (reachable, visible) elements of the data type. Let $P_v : s \rightarrow \texttt{bool}$ be a predicate that describes the visible elements of sort s, let $op : s_1 \times s_2 \times \cdots \times s_n \rightarrow s \in OP$ be an operation, then we must add the *closedness formula* $\forall x_1, x_2, \ldots x_n \bullet P_v(x_{i_1}) \wedge P_v(x_{i_2}) \wedge \ldots P_v(x_{i_k}) \implies P_v(op(x_1, x_2, \ldots, x_n))$ to the relational formulas of the implementation unit (where $i_1, i_2, \ldots, i_k$ are the indices of argument sorts with $s_{i_j} = s$).

In general it is not feasible to automatically derive an appropriate predicate P_v. Thus, *we demand that the user provides a definition for the visibility predicate.* The construction of the closedness formulas is then straightforward.

If the sort is declared to be a free type, we know that the elements consist of exactly those values that can be constructed with the help of constructor operations. In this case, an automatic construction of P_v is possible.

Multiple Representations The implementation of functions must not distinguish between different representations of the same (interface) value. Let s be the implemented sort, let $\approx_s$ be an equivalence relation, such that $t \approx_s t'$, iff t and t' are representations of the same value, let $op : s \rightarrow s \in OP$ be an operation, then we must add the *congruence formula* $\forall x, x' \bullet (x \approx_s x') \implies op(x) \approx_s op(x')$ to the relational formulas of the implementation unit. If the target sort is different, we replace $\approx_s$ with $\equiv : \forall x, x' \bullet (x \approx_s x') \implies op(x) \equiv op(x')$.

The generalization to several variables is obvious: if the ith argument sort is s, x_i and x'_i must be equivalent, i. e. $x_i \approx_s x'_i$, otherwise both must be equal, i. e. $x_i \equiv x'_i$.

Again, it is not feasible to let the compiler derive automatically the equivalence relation. *We demand that the user provides a definition for the equivalence relation*, if the data-type implementation uses multiple representations.

3.3.7 The Interpretation Relation

The implementation relation presented in Section 3.3.4 is not the only way to represent development by stepwise refinement. In this section we treat the *interpretation relation* as defined in the SPECWARE manual [JSB+95]. The interpretation relation is used for development in the SPECWARE language SLANG.

Let $\mathcal{S}$ and $\mathcal{S}'$ be two specifications. An *interpretation from $\mathcal{S}$ into $\mathcal{S}'$* consists of three components:

- A *mediator* specification $\mathcal{M}$.
- A *source morphism* $s : \mathcal{S} \rightarrow \mathcal{M}$.

- A *target morphism* $s' : \mathcal{S}' \to \mathcal{M}$ that must be a definitional extension.

The SPECWARE manual defines a definitional extension as follows. A morphism $h : \mathcal{S} \to \mathcal{S}'$ is a definitional extension, if

- h is injective, and
- every sort or operation in $\mathcal{S}' \setminus h(\mathcal{S})$ (i. e. outside the image of h) is a defined sort or operation.

A *defined sort or operation* is a sort or operation that has a definition that generates a carrier set or a "provably functional relation"[3]. In our framework, a defined sort or function is characterized by the fact that its behaviour is determined entirely by constructive formulas.

Example Since this construction is rather different from the other interpretation relations, we present a short example. We pick up the example from Program 3.1 and show an interpretation from `Colour` into `TripleReal`. Program 3.2 shows the source and the target of the interpretation.

Program 3.2 The Source and Target of the Example Interpretation

```
SIGNATURE Colour

  TYPE colour == red green blue
```

```
SIGNATURE TripleReal
  IMPORT Real COMPLETELY

  TYPE triplereal ==
       triple(1st: real,
              2nd: real, 3rd: real)
```

The mediator is shown in Program 3.3. (We omitted the discriminator functions for brevity.) In this example the type implementation is very simple. In [JSB+95] more involved examples are presented (e. g. an interpretation of sets into bags), where the intermediate type is more complex (e. g. the subtype of all bags without duplicates for the interpretation of sets into bags).

Finally, we can define the interpretation. Program 3.4 shows the definition in ad hoc pseudo code. `Colour` and `TripleReal` are the domain and codomain of the interpretation respectively, `ColourAsTriple` is declared to be the mediator specification. The source morphism s is given after the keyword DOM − TO − MED, the target morphism s' is introduced by the keyword COD − TO − MED and is declared to be the (inclusion) morphism induced by the import statements. Because the additional items in Unit `ColourAsTriple` are not only declared but also *defined* (by the DATA and DEF keywords), the import morphism is a definitional extension.

[3] According to the SPECWARE manual this property is checked syntactically.

Program 3.3 The Mediator of the Example Interpretation

```
SIGNATURE ColourAsTriple
  IMPORT Real COMPLETELY
  IMPORT TripleReal COMPLETELY

  DATA colourAsTriple == pack(unpack: triplereal)

  FUN redAsTriple greenAsTriple BlueAsTriple: colourAsTriple
  DEF redAsTriple   == pack(triple(1, 0, 0))
  DEF greenAsTriple == pack(triple(0, 1, 0))
  DEF blueAsTriple  == pack(triple(0, 0, 1))
  ...
```

Program 3.4 The Definition of the Example Interpretation

```
INTERPRETATION ColourIntoTripleReal: Colour ⟹ TripleReal
 MEDIATOR ColourAsTriple
 DOM − TO − MED colour → colourAsTriple
                red    → redAsTriple
                green  → greenAsTriple
                blue   → blueAsTriple
                ...
 COD − TO − MED IMPORT MORPHISM
```

Note that the introduction of a separate "INTERPRETATION" unit is not appropriate for our framework. We do not want to introduce additional types of units. So we would have to add the morphisms to one of the existing units. The best choice is the mediator specification, because it is specific to the respective interpretation.

Hence, the correctness conditions for the specification morphisms are part of the correctness conditions of the mediator specification[4]. The result is that the formulas translated by the morphism are added as relational formulas to the mediator specification. This construction is simpler than the application of $\mathcal{F}^{-1}$ for the implementation relation.

Correctness We must discuss whether the interpretation relation is suitable for a consistency proof. Suppose we have two units with specifications $\mathcal{S}$ and $\mathcal{S}'$ such that there exists an interpretation from $\mathcal{S}$ into $\mathcal{S}'$ with mediator $\mathcal{M}$, source morphism $s : \mathcal{S} \to \mathcal{M}$, and target morphism $s' : \mathcal{S}' \to \mathcal{M}$.

Suppose $\mathcal{S}'$ is consistent. Then there exists an algebra $A_{\mathcal{S}'}$ that is a model of $\mathcal{S}'$. Since the target morphism s' is a definitional extension, we can use it to

[4]The proof obligations entailed by a specification morphism are dealt with in Section 4.2.3.

construct an algebra $A_{\mathcal{M}}$ as follows (where m is a sort or operation from $\mathcal{M}$). For the elements that are in the image of s' the morphism is an inclusion and we can use the corresponding carrier set or operation from the algebra $A_{\mathcal{S}'}$. The other elements are defined sorts or operations, so we can derive a definition for the carrier set or the operation – this derivation is called "impl" in the following formula. Note that the inverse $s'^{(-1)}(m)$ is defined if $m \in \text{img}(s')$, because m is injective.

$$A_{\mathcal{M}}(m) = \begin{cases} A_{\mathcal{S}'}(s'^{(-1)}(m)), \text{if } m \in \text{img}(s') \\ \text{impl}(m), \text{otherwise} \end{cases}$$

We can finally apply the forgetful functor $\mathcal{V}_s$ to the algebra $A_{\mathcal{M}}$ and get the algebra $\mathcal{V}_s(A_{\mathcal{M}})$, which is a model of $\mathcal{S}$, i. e. $\mathcal{V}_s(A_{\mathcal{M}}) \in Alg_{\mathcal{S}}$. The construction is shown graphically in Figure 3.1.

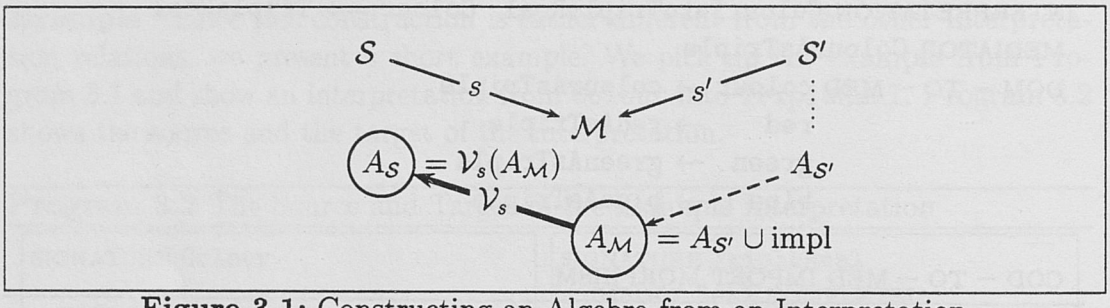

Figure 3.1: Constructing an Algebra from an Interpretation

As a result, we arrive at the conclusion that the interpretation relation can be integrated into our framework. It is possible to perform a proof of consistency by establishing an interpretation relation from a specification $\mathcal{S}$ into a consistent specification $\mathcal{S}'$.

3.4 Constructing a Correctness Proof

In the previous sections we have discussed different methods to feasibly prove consistency. In this section we go one step further and describe abstract algorithms to actually prove the correctness of a unit.

3.4.1 Computing Proof Obligations

In Section 3.2.3 we have defined a syntactic and a semantic classification of formulas. Up to now, we have classified formulas according to the syntactic scheme. Now we must classify the formulas as *axioms* or *theorems*. The validity of axioms is guaranteed by one of the methods presented in Section 3.3. So we must prove

that formulas that are classified as theorems are indeed theorems: $F_{AX} \implies F_{TH}$. Since we have to provide proofs for these formulas, we call the theorems also *proof obligations.*

Constructive Formulas Constructive formulas always hold by definition and therefore are always axioms (Section 3.2.3).

Algebraic Formulas If the proof is performed by model construction, the algebraic formulas are theorems (Section 3.3.1). If the proof is done by algebraic relation, we do not know which formulas are axioms and which are theorems. A safe default, which minimizes the number of proof obligations, is to assume that all algebraic formulas are axioms.

External Formulas External formulas are always axioms; this is explained in Section 4.1.3.

Relational Formulas Relational formulas are always theorems (Section 3.3.2).

The table in Figure 3.2 shows how the formulas are partitioned into axioms and theorems.

	syntactic category			
proof method	F_C	F_A	F_E	F_R
proof by model construction	axioms	theorems	axioms	theorems
proof by algebraic relation	axioms	axioms	axioms	theorems

Figure 3.2: Computing Proof Obligations

Extent of Justification We have suggested in Section 1.2.2 that the user be able to decide to what extent the correctness of the software system should be checked.

Once the proof obligations have been computed, the compiler can apply additional filters to select only those proof obligations that the user is interested in, e. g. only definedness conditions.

In the OPAL/J prototype we decided to mark the user's choice by a compiler directive (pragma `/$ PROOFCHECK $/`) in the source code. This decision sometimes caused confusion and hinders an easy change of the desired extent of justification. It seems preferable to introduce compiler options for this purpose and not to record the extent of justification in the source code at all.

3.4.2 Structuring the Proof of Correctness

If theorem-provers were more powerful, we could give $\bigwedge F_{AX} \implies \bigwedge F_{TH}$ as an input to a theorem-prover and use its output to determine the consistency of the unit. There are several problems that make this approach non-feasible. Testing and program synthesis cannot be used to justify arbitrary formulas, but are (in general) restricted to specifications of functions. And the efficiency of theorem-provers depends on the number of formulas, which is large in real-life applications.

Since the formula $\bigwedge F_{AX} \implies \bigwedge F_{TH}$ is not suitable for most of our envisaged justification methods, we split the proof of correctness into smaller proofs that can be justified with a user-chosen method. Of course, we must do this in a way that enables the compiler to reconstruct the correctness proof. If the following conclusions are valid in the respective institution, we can proceed:

$$(\Gamma \implies \Delta \text{ and } \Gamma \implies \Delta') \implies (\Gamma \implies \Delta \cup \Delta') \quad (1)$$

$$(\Gamma \implies \Delta) \implies (\Gamma \cup \Gamma' \implies \Delta) \quad (2)$$

$$(\Gamma \implies \{\phi\} \text{ and } \Gamma \cup \{\phi\} \implies \Delta) \implies (\Gamma \implies \Delta) \quad (3)$$

Property (1) is the most important one, because it allows to prove each proof obligation separately. Property (2) allows to restrict the set of premises, and is important for efficiency. Property (3) allows to use previously justified proof obligations.

3.4.3 Proof Declarations

Property (1) allows the separate justification of proof obligations. We go one step further and require that the user ***declares for each proof obligation separately*** the axioms that are used to prove the respective proof obligation. These ***proof declarations are part of the source code***, because they reflect the user's reasoning why the program is correct. The justification itself is part of the source code as well. Hence, proof declaration and justification form a pair of connected entities, similar to function declaration and function definition. Both entities are joined by a name.

The general shape of a proof declaration is $PD : \{f_1, f_2, \ldots, f_n\} \implies f$, where PD is the name of the proof declaration, the f_i are the *premises* and f is the conclusion or the *target* of the proof declaration.

The introduction of proof declarations poses additional work on both the programmer and the compiler writer. The compiler writer must deal with additional syntactic entities that may contain syntax and context errors on their own and must be integrated into the abstract syntax tree. The programmer on the other

hand must be aware of the proof obligations of the program unit and add for each proof obligation the "reason" why it is correct.

The structuring of the correctness proof is helpful in situations where the unit is *not* correct. It is often difficult to find out why a formal proof did not succeed. Proof declarations effectively structure the proof of correctness of the whole unit. Errors can be assigned to a specific justification, which makes debugging of justifications easier. The accompanying justifications can be checked independently, possibly in parallel, because all dependencies are contained in the proof declarations, which are checked in a separate step.

3.4.4 Context Conditions

The introduction of proof declarations introduces additional context conditions. The compiler must check the additional context condition that the proof declarations may be combined by Properties (1)-(3) to a correctness proof.

In Figure 3.3 the context conditions are summarized.

Completeness	For every proof obligation ob_i there must exist a proof declaration $P : \Gamma \vdash ob_i$
Non-circularity	There exists an ordering of proof obligations, such that each proof declaration with target ob_i only uses proof obligations $ob_1, ob_2, \ldots, ob_{i-1}$ in its premises

Figure 3.3: Context Conditions for Proof Declarations

Completeness and non-circularity are easily checked automatically. It is even possible to let the compiler find out whether a non-circular ordering exists. (Actually, there are similar context checks necessary in the OPAL compiler.)

3.4.5 Justifications

The proof declarations form the interface between the formal algebraic definition of correctness and the less formal justification methods. The proof declarations provide a standard way of integrating different justification methods.

The correctness check for a unit requires that each proof declaration has an associated justification and that ***these justifications all succeed.***

We treat justifications thoroughly in Section 6.

Chapter 4

Modular Correctness

Example programs are often very small, but in reality software projects consist of many *components*, which are put together in various ways to form the final product. The import operator is the most basic composition operator that is available in virtually all programming languages. Functional programming languages employ some kind of polymorphism, and some algebraic specification languages have other operations like quotient and subalgebra construction.

In the following section we discuss some general principles for modular correctness. After that we study various operations used to construct new units on the basis of other units. These operations include operations that are not generally found in programming languages as individual operations. Breaking up complicated operations into simpler parts eases explanation of the resulting proof obligations. In the examples we had to invent an ad hoc syntax for operations that are not included in OPAL or the currently discussed OPAL 2α.

4.1 General Principles of Modular Correctness

In the treatment and the selection of composition operators we have to respect some common principles. These principles are derived from the need to use these composition for functional programming with correctness justifications.

4.1.1 Units Are Independent

Units are developed *independently*. The use of a unit `V` for the construction of another unit `U` depends only on the interface of `V` and not on the implementation of `V`. In our framework the interface and the implementation are independent

units that are connected via an implementation relation. So we need to restate the usual formulation, and demand that using a unit V does not depend on any of the units that implement this unit[1].

This raises the question how the independent development carries over to the proof (or rather justification) of correctness. Suppose, for example, that a unit U uses (e. g. imports) a unit V; see Figure 4.1. The correctness proof of unit V might be deferred, because it shall be done by providing an implementation, which is not yet available. So unit V has not been proven correct, but what about unit U? If we demand that V is proven correct, before we can prove the correctness of U, the correctness of U depends on the implementation V'. It is undesirable that the compilation with correctness check of unit U depends on an implementation for V whereas the "classic" compilation without correctness check does not.

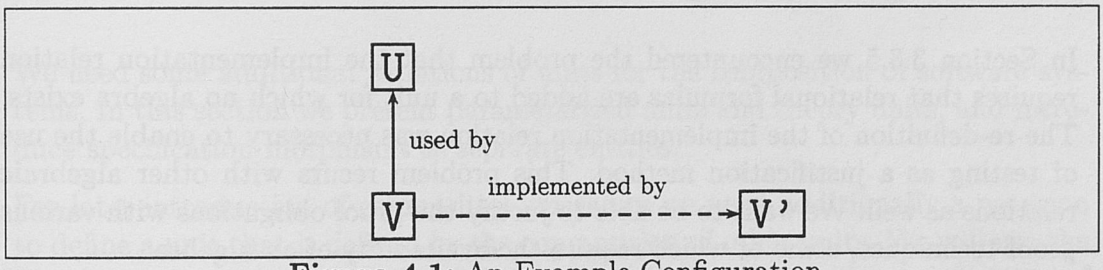

Figure 4.1: An Example Configuration

We suggest to distinguish between *"global correctness"* and *"relative correctness"*. There is no question that the whole software system is only correct if every unit is correct. We call this "global correctness". But during the development of a unit U, we will perform the correctness proof under the assumption that all units $V_1, V_2, \ldots$ used in the implementation of U are correct. The correctness proof actually performed is thus weakened and now reads "If all of the units $V_1, V_2, \ldots$ are correct, then so is unit U." This weaker form is what we call "relative correctness".

Actually, the check for global correctness is out of scope for a compiler. The compiler is concerned with a single unit and has no knowledge about the system as a whole. The task of checking the global correctness is better assigned to the *linker*. The linker should check whether all units the software system is composed of have been proven relative correct. Even if the difference between linker and compiler is concealed by an integrated development environment, it is useful to separate between the treatment of a single unit and the building of a software system.

The differences between global and relative correctness are summarized in the following table:

[1] If we adopt the notion of "horizontal" and "vertical" structuring, this translates to a real principle of orthogonality.

	checked at	*checked entity*	*correctness*
global correctness	link time	whole software system	absolute
relative correctness	compile time	single unit	relative to units used

The distinction of relative and global correctness makes the independent development of units together with their respective correctness proofs possible, and thus improves the integration of correctness proofs into the software development process.

4.1.2 Availability of an Algebra

In Section 3.3.5 we encountered the problem that the implementation relation requires that relational formulas are added to a unit for which no algebra exists. The re-definition of the implementation relation was necessary to enable the use of testing as a justification method. This problem recurs with other algebraic relations as well. We want to be able to justify the proof obligations with various proof techniques, some of which require the availability of an algebra.

The only way we have to construct an algebra is provided by the compiler, which uses the constructive formulas to construct an algebra. We do not have the possibility to change an existing algebra. (It might be possible to automatically derive definitional equations, but this is a meta-operation on source-code level.) Therefore we have to limit the specification-building operations to those operations that *protect* (i. e., do not change) the argument algebra.

4.1.3 External Formulas

We have already introduced the notion of "external formulas" but have not yet made use of them. The concept of relative correctness is the reason for the introduction of the category of external formulas. Since correctness of a unit `U` with semantics $\mathcal{S} = (S, OP, F)$ is proven under the assumption that every used unit `V` is correct, we can safely assume that formulas of `V` are valid. It is the "responsibility" of `V` (or rather the developer of `V`) to show that the formulas of `V` are valid. The developer of `U` may use formulas from `V` in the correctness proof of `U`.

The correctness proof by constructing a model is only slightly changed. External formulas are treated as axioms in the correctness proof of `U`: $F_C \cup F_E \implies F_A$.

The correctness proof by implementation is in principle not changed. In the general case, we must prove $F' \implies \mathcal{F}^{-1}(F)$. We can omit the relational formulas

of F, because these are always theorems. With external formulas, this reads $F'_C \cup F'_E \cup F'_A \implies \mathcal{F}^{-1}(F_C) \cup \mathcal{F}^{-1}(F_E) \cup \mathcal{F}^{-1}(F_A)$. The implementation relation affects only elements that are introduced in one of the related units. For external formulas the functor $\mathcal{F}$ and therefore also the inverse translation $\mathcal{F}^{-1}$ is the identity. Quite often we will have the situation that $F_E \subseteq F_{E'}$; the OPAL language, for example, enforces this relationship. In this case, there is no need to prove the translated external formulas and the proof obligation simplifies to

$$\underbrace{F'_C \cup F'_E \cup F'_A}_{F'} \implies \mathcal{F}^{-1}(F_C) \cup \mathcal{F}^{-1}(F_A)$$

4.2 Entities for Modular Programming

We need some additional variations of units for the composition of software systems. In this section we present parameterized units and theory units, and introduce specification morphisms as separate entities.

For later introduction of composition operators we need additionally a notation to define a unit that is defined by the composition of other units. We will use the notation "STRUCTURE U IS $V_1 \circ V_2$" to denote that unit U is the composition of V_1 and V_2 by the composition operator $\circ$.

4.2.1 Parameterization

Parameterization is an important technique for code re-use, and can also be used on the unit level. A certain part of the unit is marked as the (formal) parameter and can be replaced by entities from another unit as the actual parameter. The mechanism for instantiation is presented in Section 4.3.3.

From the algebraic point of view a parameterized specification is a pair of specifications, one of which (the parameter) is included in the other (the body) $PAR \hookrightarrow BODY$. The fact that the parameter is again a (closed) specification poses some problems. Quite often, only a part of the parameter is meant to be freely instantiated. The "formal bool" problem [Cla91] was discovered when the first version of ACT ONE was introduced as the specification language of the LOTOS environment. Since `bool` is part of nearly every specification, it is part of almost every non-trivial formal parameter specification, and could be instantiated with other sorts.

Later versions of ACT ONE ([Did92], [CEW93]) introduce additional language elements to distinguish the part of the parameter that can be freely instantiated from the part of the parameter that is fixed. Figure 4.2 shows the resulting

situation. Note that the sets of sorts, operations and formulas that are designated with a superscript "0" do not constitute a closed specification. In functional programming languages, most often only the changeable part of the parameter (S_P^0, OP_P^0, F_P^0) must be explicitly designated as the parameter.

$$\underbrace{(S_F, OP_F, F_F) \subseteq (S_P, OP_P, F_P)}_{\text{parameter}} \subseteq \underbrace{(S_B, OP_B, F_B)}_{\text{body}}$$

S_F	OP_F	F_F	fixed part of parameter
S_F^0	OP_F^0	F_F^0	changeable part of param.
$S_P = S_P^0 \cup S_F$	$OP_P = OP_P^0 \cup OP_F$	$F_P = F_P^0 \cup F_F$	parameter
$S_B = S_B^0 \cup S_P$	$OP_B = OP_B^0 \cup OP_P$	$F_B = F_B^0 \cup F_P$	body

Figure 4.2: The Algebraic View on Parameterization

A specification morphism m between parameterized specifications $(PAR_1 \hookrightarrow BODY_1)$ and $(PAR_2 \hookrightarrow BODY_2)$ is a specification morphism $m : BODY_1 \to BODY_2$ that repects the parameter, i. e. PAR_1 is mapped to PAR_2.

Program 4.1 shows (part of) a parameterized unit that will be used in the following examples. The sort `bool` is part of the parameter specification, but not part of the variable part. It is introduced in the unit `BOOL`, which is imported into every unit. The part of the parameter that can be freely instantiated consists of only those entities that are introduced in the parameter specification.

Program 4.1 An Example for a Parameterized Unit

```
SIGNATURE DataWithOrd
  SORT α
  FUN ◁: α × α → bool

  LAW dfd          == ALL x y. DFD x ◁ y
  LAW total        == ALL x y. x ◁ y OR y ◁ x OR x ≡ y
  LAW transitive   == ALL x y z. x ◁ y AND y ◁ z ⟹ x ◁ z
  LAW irreflexive  == ALL x. NOT x ◁ x
SIGNATURE Set

  PARAM DataWithOrd

  SORT set
  FUN {}: set
  FUN incl: α × set → set
  FUN in: α × set → bool
  ...
```

We must decide whether the formulas of the parameter part belong to the axioms or to the theorems. Parameterized specifications as such (uninstantiated) cannot be used in a software system, it is always necessary to instantiate them. The actual parameter unit contains a justification for the parameter formulas, otherwise the correctness proof of the actualization morphism fails. Hence, there is no need to prove the properties in F_P, and we classify them all as belonging to the external formulas of the parameterized unit.

4.2.2 Algebraic Relations and Parameterized Units

The introduction of parameterized units requires an adaptation of the definition of proof obligations for algebraic relations.

Equivalence We must not only show that the bodies are equivalent, we must also show that the parameter parts are equivalent. This results in additional relational formulas: Let $\mathcal{S}$ be the specification that is known to be correct, let $\mathcal{S}'$ be the specification that is supposedly equivalent, let $F_P = \{f_1, f_2, \ldots, f_n\}$ and $F'_P = \{f'_1, f'_2, \ldots, f'_m\}$ be the formulas of the parameters of the related units, then the relational formulas are $m^*_{\leftrightarrow}(\bigwedge F_P) \implies f'_j$ for $j = 1, 2, \ldots, m$ and $\bigwedge F'_P \implies m^*_{\leftrightarrow}(f_i)$ for $i = 1, 2, \ldots, n$, where $m^*_{\leftrightarrow}$ is the continuation of the isomorphism $m_{\leftrightarrow} : \mathcal{S} \to \mathcal{S}'$ between the two specifications.

Implementation The implementation relation in general requires that the implementation unit has fewer models than the implemented unit. For the parameter, this condition is reversed, though. It is acceptable, if the implementation can be done with a parameter that is less restricted than the original parameter. The definition of implementation for parameterized units is changed to (compare this to the definition in Section 3.3.5):

($\mathcal{S}$ is implemented by $\mathcal{S}'$) **iff**

$$(\mathcal{F}^{-1}(F_P) \implies F'_P) \wedge ({F'}^0_B \implies \mathcal{F}^{-1}(F^0_B)) \wedge (Alg_{\mathcal{S}'} \neq \emptyset)$$

If the parameter of both units is equal, we can ignore the parameter formulas in the construction of the relational formulas. OPAL enforces this condition, so we can make use of this simplification for OPAL/J.

Interpretation For the interpretation relation nothing really changes. We must prove that the source and target morphism given are indeed specification morphisms. This is a little bit more complicated for parameterized specifications than in the standard case.

4.2.3 Specification Morphisms

Specification morphisms are central to the algebraic treatment of parameterization, but also used for renaming. Most often specification morphisms are implicitly denoted, sometimes specification morphisms are explicitly given, but they are not given the status of "citizens", let alone "first-class citizens". OBJ, ACT ONE-$\underline{\subset}$, and SPECWARE are exceptions of this rule.

We introduce here specification morphisms as separate entities in order to exhibit the proof obligations that are induced by the declaration of a specification morphism.

Program 4.2 A Specification Morphism

SPECIFICATION MORPHISM `m`

```
FROM DataWithOrd TO Nat BY α                  → nat
                          ◁'DataWithOrd       → >' Nat
```

We assume that items that are not introduced in the source specification (e. g. the ubiquitious `bool`) are mapped to the corresponding items in the target specification by default.

We recall from Section 3.1 that a specification morphism is a signature morphism that respects the formulas, i. e. $F_2 \implies m^*(F_1)$. This directly translates to the proof obligations for the correctness of the specification morphism `m`. We have to prove the validity of the translated laws `dfd`*, `total`*, `transitive`* and `irreflexive`* from the formulas of `Nat`.

Feasible Specification Morphism Specification morphisms are frequently used in the construction of software systems. If specification morphisms are declared separately, the additional proof obligations are not a big burden, but specification morphisms are often declared implicitly. In this case, the cost of the check of the proof obligations becomes important and the semantic implication "$\implies$" too expensive.

We propose to ***replace the semantic implication "$\implies$" in $F_2 \implies m^*(F_1)$ by the subset relation***. Of course, if $F_2 \supseteq m^*(F_1)$ holds, then we also have $F_2 \implies m^*(F_1)$. Hence, our proposal is sound. The advantage is that the compiler can check the proof obligations syntactically. We demand that the translated formulas of the source specification are part of the target specification. The following table compares the characteristics of the standard proof obligations and our proposal for a feasible definition of proof obligations.

	standard proof obligations	feasible proof obligations
definition	$F_2 \Longrightarrow m^*(F_1)$	$F_2 \supseteq m^*(F_1)$
correctness check	semantic	***syntactic***
cost	high	low
automatable	no	yes

Program 4.3 shows the adapted declaration of the specification morphism. (We assume that `Nat` contains appropriate formulas for the > function.)

Program 4.3 The Specification Morphism – Improved Version

```
SPECIFICATION MORPHISM m

FROM DataWithOrd TO Nat BY α          → nat
                            <' Data   → >' Nat
                            dfd       → dfd_gt
                            total     → total_gt
                            transitive → transitive_gt
                            irreflexive → irreflexive_gt
```

The source code of Program 4.3 is longer than the source code of Program 4.2, so the introduction of feasible proof obligations does not really look like an improvement. The introduction of theories in the next section solves this problem.

The feasible proof obligations are better suited for implicit specification morphisms, which should not be burdened with heavy proof obligations. Of course, the validity of laws like `dfd_gt`, `total_gt`, `transitive_gt` and `irreflexive_gt` must be proven in the context of unit `Nat`, i. e. in the context of the target specification of the morphism. So we have a "principle of conservation of energy": *somewhere* the validity has to be proven.

For explicitly defined specification morphisms the feasible proof obligations are too restrictive. It is not necessary to avoid proof obligations that must be checked semantically, because we have a separate unit where we can attach these proofs to.

The experience with OPAL/J shows that the feasible proof obligations fit well in the existing environment for the following reasons:

- Specification morphisms are perceived as simple objects and should not be burdened with separate proof obligations.
- On the other hand, units are regarded as complex and the addition of further proof obligations is acceptable.

- Specification morphisms often require the proof of simple mathematical properties that are anyway part of the specification. So it is often no extra work for the developer to explicitly denote the properties used in actualization morphisms.
- The compiler can easily check the correctness of the specification morphisms and provide useful error messages, if the proof obligations for an actualization morphism are not fulfilled.

The introduction of theories in the following section further facilitates the check of specification morphisms for the compiler and programmer.

4.2.4 Theories

The introduction of laws into functional programming languages requires the introduction of an additional type of unit. Theories are introduced (e. g. into OPAL 2α or OBJ) to group properties together into a single unit and apply these properties to groups of functions. Program 4.4 shows the theory of total orders as it could have been used in Program 4.1.

Program 4.4 The Theory of Total Orders

```
THEORY TotalOrder
  SORT α
  FUN ◁: α × α → bool

  LAW dfd        == ALL x y. DFD x ◁ y
  LAW total      == ALL x y. x ◁ y OR y ◁ x OR x ≡ y
  LAW transitive == ALL x y z. x ◁ y AND y ◁ z ⟹ x ◁ z
  LAW irreflexive == ALL x. NOT x ◁ x
```

A first application of theories is to shorten the denotation of specification morphisms with explicit formulas. Suppose that `DataWithOrd` and `Nat` contain the necessary declarations (see Program 4.5[2]), then the specification morphism `m` can be written as shown in Program 4.6.

The compiler can easily find the correspondences for the laws of the theory, which therefore need not be written down explicitly. The syntactical representation of the specification morphism is again as short as the first variation shown in Program 4.2. This time, however, the correctness of the specification morphism has already been proven by the compiler.

[2]The keyword ASSERT is introduced in Section 4.3.5.

Program 4.5 DataWithOrd and Nat Using Theories

SIGNATURE DataWithOrd

SORT α
FUN $\lhd$: $\alpha \times \alpha \to$ bool

ASSERT TotalOrder

SIGNATURE Nat

TYPE nat == 0
succ(pred: nat)

FUN < = >: nat × nat → bool
ASSERT TotalOrder
RENAMED BY $\alpha \to$ nat
$\lhd \to >$

...

Program 4.6 The Specification Morphism Using Theories

SPECIFICATION MORPHISM m

FROM DataWithOrd TO Nat

BY α	$\to$	nat	*optional*
$\lhd'$Data	$\to$	$>'$ Nat	
dfd[$\alpha, \lhd$]	$\to$	dfd[nat, >]	
total[$\alpha, \lhd$]	$\to$	total[nat, >]	
transitive[$\alpha, \lhd$]	$\to$	transitive[nat, >]	
irreflexive[$\alpha, \lhd$]	$\to$	irreflexive[nat, >]	

The main reason for the introduction of theories is that we need a different semantics for theories and standard units. Theories are not intended to be translated to executable code. Theories are introduced to be able to add algebraic properties to sorts and functions introduced in standard units, and we want this intent to be reflected in their semantics.

A non-standard, but elegant semantics for theories is the "classical" or "hyperloose" semantics proposed in [Pep91]. By this semantics every algebra that has a subalgebra isomorphic to an $Alg_{\mathcal{S}}$-algebra is considered an element of the hyperloose semantics. More precisely, let $\mathcal{S} = (S, OP, F)$ be the base specification contained in the source code of the theory. Then we have

$$HLAlg_{\mathcal{S}} = \bigcup_{\mathcal{T} \in SPEC} \{A \in Alg_{\mathcal{T}} \mid \exists A' \subseteq A \bullet A' \in Alg_{\mathcal{S}}\}$$

The correctness of a theory can be similarly defined to that of a standard unit as consistency, i. e. non-emptyness of the semantics. Both methods – proof by algebraic relation and proof by providing a model – are possible. Because theories are not intended to be given a concrete implementation, we suggest a different syntax for proof by constructing a model. Instead of providing an implementation

in a different unit, our idea is to require the user to declare a *witness* within the source code of the theory, see Program 4.7.

Program 4.7 The Theory of Total Orders with Witness

```
THEORY TotalOrder

  SORT α
  FUN ◁: α × α → bool

  LAW dfd          == ALL x y. DFD x ◁ y
  LAW total        == ALL x y. x ◁ y OR y ◁ x OR x ≡ y
  LAW transitive   == ALL x y z. x ◁ y AND y ◁ z ⟹ x ◁ z
  LAW irreflexive  == ALL x. NOT x ◁ x

  WITNESS Nat ONLY nat <
          RENAMED BY nat → α  <→ ◁
  ...
```

The identifiers after the ONLY keyword[3] constitute the subalgebra of `Nat` that must be proven to belong to the theory. If the correctness of a theory is to be proven by a witness, the formulas of the witness are added as external formulas F_E, and the formulas introduced in the theory are treated as algebraic formulas F_A. In Figure 3.2 the partitioning of formulas into axioms and theorems for proof by model construction was shown: F_C and F_E are axioms, F_A and F_R are theorems. In a theory, we do not have constructive formulas, so we must prove $F_E \implies F_A \cup F_R$.

We will continue the discussion in Section 4.3.5. There we will introduce the assertion operation (the "import" for theories)

4.3 Import and Related Operations

Importing other units is the most important operation for combining modular software. The import operation as found in functional programming languages, such as OPAL, is in fact a compound operation, which comprises several simple algebraic operations. These are treated separately in the following, before the complex import is discussed. Finally, we introduce the assertion operator, which is the import operator for theories.

[3]The ONLY keyword is treated in Section 4.3.2.

4.3.1 Simple Import

We start with the most important and most often used operation to construct new units, namely the *import* of one unit into another. (We do not consider selective import here, but further down in a section on restriction; see Section 4.3.2.)

Let `V` be the unit that is imported into unit `U`. The semantics of `V` is $\mathcal{S}_V = (S_V, OP_V, F_V)$, the semantics of `U` contains that of `V` and is represented as $\mathcal{S}_U = (S_U, OP_U, F_U) = (S_V \cup S', OP_V \cup OP', F_V \cup F')$, where S', OP', F' denote those specification elements that are introduced in unit `U`. Note that S', OP', F' in general do not constitute a closed specification.

The compiler will use the model constructed for unit `V` unchanged in the construction of the model of `U`, in general it will not even physically copy the model of `V` but use references. This method must be mirrored by the semantics. We require therefore that the models of `U` contain a model of `V`:

$$\forall A \in Alg_{\mathcal{S}_U} \bullet A_{/\Sigma_V} \in Alg_{\mathcal{S}_V}$$

This property is ensured, if those formulas that refer to items of `V`, i. e. those formulas from F_{U/Σ_V}, are consequences of the formulas F_V. Formulas of F_{U/Σ_V} must therefore not be constructive formulas, because these are always interpreted as axioms in our approach.

The language definition of OPAL prohibits redefinitions of functions, as do the definitions of most other languages. Formulas of F_{U/Σ_V} therefore can be algebraic formulas only. So there is no difficulty in demanding that formulas from F_{U/Σ_V} must be proven from the imported axioms.

Following the principle of relative correctness, we conclude that the formulas F_V belong to the axioms, since these are external formulas with respect to `U`.

4.3.2 Restriction

With the help of restriction the developer can reduce the number of names that are visible. This helps to reduce name clashes; it is not essential for programming. Program 4.8 shows a restriction of unit `Nat` to the type `nat` and the greater function.

Program 4.8 A Restriction of `Nat`

```
STRUCTURE NatRestrict
  IS Nat ONLY nat >
```

The construction is always possible provided that the resulting specification is closed, i. e. the part after the ONLY must constitute a specification. The corresponding construction on the underlying algebra is simple: the items not mentioned in the restriction are removed, but the carrier sets and operations that are kept are unchanged and therefore fulfil the same algebraic properties as the corresponding functions in the unrestricted algebra. Formulas that contain removed names are not visible in the resulting specification.

The change of the algebra is very simple and does not prevent the use of testing as justification method. Every function from the restricted algebra can be replaced by the corresponding function from the base algebra.

4.3.3 Instantiation

In the framework presented in Chapter 3 instantiation refers to the process of replacing the formal parameters of a unit with actual parameters. For languages that support polymorphic definitions (e. g. ML), instantiation is done in a similar fashion, but separately for every sort and function.

The algebraic view on instantiation is shown in Figure 4.3. The upper row represents the parameterized unit, the lower left-hand specification is the actual parameter. The lower right-hand specification is the categorical pushout of the diagram. The dashed arrows and the pushout specification are constructed by the compiler. The user must provide the actualization (specification) morphism, the target of which defines the actual parameter.

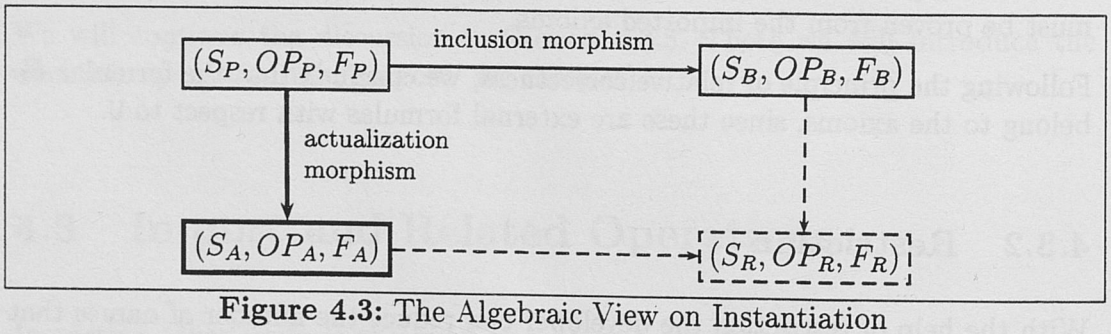

Figure 4.3: The Algebraic View on Instantiation

The actualization morphism is the central point of the construction. We continue the example from Programs 4.3 and 4.6 in Program 4.9.

Program 4.9 An Instantiation of `Set`

```
STRUCTURE NatSet
  IS Set ACTUALIZED BY m
```

Besides being a specification morphism, there are two further requirements. First, the source specification of the actualization morphism must be the same as the formal parameter. This requirement can easily be checked automatically. Second, the actualization morphism must not change the fixed part of the parameter specification. Again, this requirement is easily checked.

Instantiation with Implicit Morphism The situation changes if the specification morphism is defined implicitly, as done in Program 4.10.

Program 4.10 An Instantiation of `Set` Using an Implicit Morphism

```
STRUCTURE NatSet
  IMPORT NatRestrict COMPLETELY
  IS Set[nat, >]
```

In this case, we inherit the proof obligations from the specification morphism (as discussed in Section 4.2.3) for the complex import, i. e. we must prove that the implicitly constructed signature morphism is also a specification morphism. The additional conditions for an instantiation morphism hold by construction of the implicit morphism and need not be checked.

We have presented a feasible variation of the definition of a specification morphism, which requires the developer to prepare the properties that are necessary to check the correctness of the instantiation. The transfer of this variation to the instantiation is straightforward. Program 4.11 shows the resulting source code.

Program 4.11 An Instantiation of `Set` Using Explicit Properties

```
STRUCTURE NatSet
  IMPORT NatRestrict COMPLETELY
  IS Set[nat, >, dfd_gt, total_gt, transitive_gt, irreflexive_gt]
```

The advantage of this formulation is that the user does not have to remember lots of trivial specification morphisms; the disadvantage is that the properties must be prepared and listed among the actual parameters. If the properties are standard properties (like the ones used in the example), it is possible to use theories to help the compiler to find and match the corresponding properties.

4.3.4 Complex Import

In Section 4.3.1 we have studied the simple import of one unit. Functional programming languages do not have separate operations like the declaration of specification morphisms, restrictions, or instantiation. Instead, these operations are

merged into the simple import. OPAL does not require the user to declare a separate unit NatSet, as we did in the examples in the previous section. Instead a single import statement suffices, see Program 4.12.

Program 4.12 Importing Sets Over Natural Numbers

```
IMPORT Nat ONLY nat >
IMPORT Set[nat, >, dfd_gt, total_gt,
                   transitive_gt, irreflexive_gt] COMPLETELY
```

Restriction and instantiation do not pose special problems and do not generate proof obligations, the only problematic operation is the actualization morphism. The correctness of the actualization morphism must be checked, and as there is no separate declaration of a specification morphism, the proof obligations are connected with the corresponding import statement. The imported properties are external properties and thus belong to the axioms of the importing unit.

Transitive Import If the import is done transitively, the two situations in Program 4.13 are treated the same. This poses no problems in the simple case, but if proof obligations for anonymous specification morphisms are involved, the situation changes. Consider the example in Program 4.13 under the assumption that Set[nat, =] is not correctly instantiated.

Program 4.13 Problems With Transitive Import

```
SIGNATURE U
  IMPORT V
```

```
SIGNATURE U
  IMPORT V
  IMPORT Nat ONLY nat =
  IMPORT Set[nat, =] COMPLETELY
```

```
SIGNATURE V
  IMPORT Nat ONLY nat =
  IMPORT Set[nat, =] COMPLETELY
```

```
SIGNATURE V
  IMPORT Nat ONLY nat =
  IMPORT Set[nat, =] COMPLETELY
```

The situation on the left-hand side requires one check of the actualization morphisms, on the right-hand side two checks are necessary. The reason for this additional check is that the specification morphisms are declared implicitly (anonymously), and the compiler performs two checks for two anonymous specification morphisms.

If the check of the specification morphism succeeds, this introduces "only" an inefficiency, but if the check fails, the relative correctness for unit U is different for the two examples. On the left-hand side, unit U is relative correct, whereas it is not on the right-hand side. "Flattening" the transitive import by lifting all

imports to the top unit is therefore not possible. At least the information about the origin of the anonymous specification morphisms must be kept, such that it is possible to determine whether the correctness must be checked in the top unit. The OPAL compiler does perform a flattening of transitive imports and had to be patched to make a correct treatment of relative correct units possible in the OPAL/J prototype.

4.3.5 Assertion

In Section 4.2.4 we have introduced theories to group properties. Program 4.14 shows an excerpt of the unit of natural numbers, which uses theories to make two assertions, namely that the functions less and greater are total orders.

Program 4.14 An Assertion of Total-Order Properties

```
SIGNATURE Nat
  TYPE nat == 0
              succ(pred: nat)

  FUN < = >: nat × nat → bool
  ASSERT TotalOrder RENAMED BY α → nat ◁ →<
  ASSERT TotalOrder RENAMED BY α → nat ◁ →>
  ...
```

Another use of assertions is shown in Program 4.15. The assertion restricts the parameter of unit Set.

Program 4.15 An Assertion for a Parameter of a Parameterized Unit

```
SIGNATURE Set[α, ◁]
  SORT α
  FUN ◁: α × α → bool
  ASSERT TotalOrder

  SORT set
  FUN {}: set
  FUN incl: α × set → set
  FUN in: α × set → bool
  ...
```

The semantics of an assertion is the claim that the models of the current unit belong to the (renamed) theory. This claim is treated like any other axiom introduced in the current unit, i. e. we regard this claim as a non-constructive and non-external property.

Recall the definition of the semantics of a theory with base semantics $\mathcal{S}$: $HLAlg_{\mathcal{S}} = \bigcup_{T \in SPEC}\{A \in Alg_T | \exists A' \subseteq A \bullet A' \in Alg_{\mathcal{S}}\}$. If we want to prove that $A \in HLAlg_{\mathcal{S}}$, we have to find a subalgebra A' of A that fulfils the formulas of the theory.

The renaming morphism already identifies the subalgebra to be used, namely the target of the renaming morphism. So we must prove that the properties of the theory hold for the indicated subalgebra.

We add the formulas of theory to the algebraic formulas of the semantics. This has the desired effect for the correctness proofs. If the correctness of the current unit is to be proven by an implementation, the properties of the theory are treated as axioms; if the correctness is proven by a model, the properties are treated as theorems.

If the asserted theory restricts the parameter, i. e. the signature of the theory is a subsignature of the parameter signature (after renaming), the formulas are not added to the algebraic formulas, but are treated as all other parameter formulas, and added to the external formulas of the unit.

4.3.6 Summary

The table in Figure 4.4 summarizes how formulas that are introduced by one of the previous operations are classified in the new unit.

operation	classification	comments
simple import	external formulas	
restriction	external formulas	only visible formulas
instantiation	relational formulas	as necessary for the actualization morphism; syntactic check
complex import	rel. and ext. formulas	combination of the three above operations
assertion	algebraic formulas	body formulas
	external formulas	parameter formulas

Figure 4.4: The Classification of Imported Formulas

4.4 Miscellaneous Operations

This section contains some operations that are not found as widespread as those previously presented. The common characteristic of these operations is that they

generate a new specification from an argument specification, whereas the other operations combine existing specifications.

It is often difficult or impossible to use the algebra of the argument specification for the new specification. Therefore an integration of these operations into our approach is often impossible, sometimes difficult, and depends on additional application conditions.

4.4.1 Renaming

Renaming is an operation that is not often found in functional programming languages but that is part of most algebraic specification languages.

The example renaming in Program 4.16 shows why we cannot unconditionally introduce the renaming operation.

Program 4.16 Natural Numbers – Wrongly Renamed

```
STRUCTURE NatRenamed
  IS Nat RENAMED BY nat → num  <→ foo  >→ foo
```

Both the less and the greater function are given the same name. For algebraic specifications, it is no problem to construct the resulting specification. However, it is not possible to construct a model for the resulting specification from the model of the source algebra.

If the renaming morphism is *bijective*, however, there are no difficulties (see Program 4.17). The existing model can be re-used for the renamed specification, so there is no need to prove the correctness of the renamed unit again.

Program 4.17 Natural Numbers – Correctly Renamed

```
STRUCTURE NatRenamed
  IS Nat RENAMED BY nat → num  <→ greater  >→ less
```

4.4.2 Extension

In algebraic specification languages, the term "extension" is often used where programming languages use "import" or "inclusion". However, there are different variants of extension. OBJ distinguishes four import modes and allows the user to chose between simple (two variants), consistent and conservative extension.

By a (simple) extension new sorts S', operations OP' or formulas F' can be added to a specification $\mathcal{S} = (S, OP, F)$, resulting in a new specification $\mathcal{S}' =$

$(S \cup S', OP \cup OP', F \cup F')$. Note that S', OP', F' need not be a closed specification. A consistent extension is a simple extension that does not identify data items from S, a conservative extension is consistent extension that does not introduce new elements to S.

- A simple extension works like the *textual inclusion* of some older programming languages like C or PASCAL. If the extension operator is used to handle modularized software systems, the compiler does not need to handle modularization itself. Extension is therefore a quick (and dirty) way to add modules to a language. But extension is an inefficient way to implement modularization. Every time the unit is compiled, the inclusion must be expanded, and the resulting (large) text is fed to the following compiler phases.

  Concerning the semantics, the unrestricted possibility to add new formulas allows to change any possible semantics of the included unit in an unforeseeable way. This prohibits any systematic treatment of extension, and makes extension an unusable specification building operation in our framework.

- A consistent extension does not identify elements, but may add new elements to S. Therefore we cannot use an algebra $A \in Alg_S$ to construct an algebra $A' \in Alg_{S'}$. In particular, we cannot extend the operations of OP to handle the new elements properly.

- For a conservative extension, we have $A \in Alg_{S'} \implies A_{/S} \in Alg_S$. Hence, a conservative extension can be expressed as a simple import. This operator was discussed in Section 4.3.1.

4.4.3 Subalgebra

The subalgebra construction restricts the carrier sets while the functions are the same as in the original algebra. Program 4.18 shows as an example a subalgebra built over natural numbers. The carrier set restriction is given by a predicate odd?, which selects the odd natural numbers.

Program 4.18 A Subalgebra of `Nat`

```
STRUCTURE NatSubalgebra
  IS Nat RESTRICT nat BY odd? ▪
```

The subalgebra construction requires to prove that the functions are closed on the restricted set. The restriction-by predicate facilitates the justification, but the justification must be performed for every function, which might be an

extensive task[4].Testing of the subalgebra property is possible, if the predicate is executable. Once the subalgebra property has been proven, the model for the original algebra can serve as a model for the subalgebra.

In general, formulas that are valid in the original specification need not hold in the subalgebra, because their validity could depend on elements that are removed in the subalgebra. Therefore, the validity of all formulas must be proven for the restricted carrier sets again.

4.4.4 Quotient

The quotient construction identifies elements by an equivalence relation. We might construct sets from lists as shown in Program 4.19.

Program 4.19 A Quotient of Seq

```
STRUCTURE SetsFromSeq
  IS Seq QUOTIENT BY equiv
  FUN equiv: seq × seq → bool

  LAW seq_as_set_1 == ALL a b S. equiv(a::b::S, b::a::S)
  LAW seq_as_set_2 == ALL a S. equiv(a::a::S, a::S)
  ...
```

The difficulties are similar to those of the subalgebra construction. However, the check that the function indeed has the equivalence property is (possibly) not as expensive as the check of arbitrarily many functions. Again the resulting specification need not be consistent, even if the original specification is. Suppose, for example, that the unit Seq contains the usual definitions for the length function (denoted #). Then the laws of Program 4.19 allow to conclude that $1 \equiv \mathtt{succ}(1)$:

$1 \rightsquigarrow \#(\mathtt{a}::\Diamond) \rightsquigarrow \#(\mathtt{a}::\mathtt{a}::\Diamond) \rightsquigarrow \mathtt{succ}(\#(\mathtt{a}::\Diamond)) \rightsquigarrow \mathtt{succ}(1)$

Therefore we must prove that the formulas of the argument specification hold for the equivalence classes.

4.4.5 Comparison with Data-Type Implementation

The subalgebra and the quotient operation as presented in the previous sections are closely related to the restriction functor and the identify functor used in the definition of the implementation relation (see Section 3.3.4).

[4]It will fail for the example in Program 4.18, because the sum of two odd numbers is even.

- Actually the restriction functor does construct a subalgebra, namely the smallest subalgebra that contains the implemented algebra. The visibility predicate demanded in Section 3.3.6 for computing the proof obligations for data-type implementation corresponds to the RESTRICT predicate in the subalgebra construction.
- Just the same the identify functor constructs a quotient algebra, namely the quotient wrt. the congruence induced by F. The QUOTIENT relation corresponds to the equivalence predicate demanded for computing the proof obligations in Section 3.3.6.

The main difference is that the subalgebra and the quotient operator *construct a new algebra* A' from an input algebra A, whereas for the data-type implementation we only *check* that A is a subalgebra (resp. a quotient algebra) of another algebra A'.

As regards program development, the difference is that the subalgebra and the quotient operator reduce the cardinality of the carrier sets, whereas the implementation increases the cardinality of the carrier sets (by junk elements or multiple representations).

Chapter 5

Integrating Justification Support

It is our aim to integrate the justification support into the classical compilation process. Figure 5.1 roughly depicts the stages of the compilation process: the parser turns source code into an internal representation as an abstract syntax tree, the context checker not only checks context conditions, but also adds attributes used in later stages. The optimizer rearranges the syntax tree in order to make the code smaller and/or faster. The coder finally synthesizes object code from the abstract syntax.

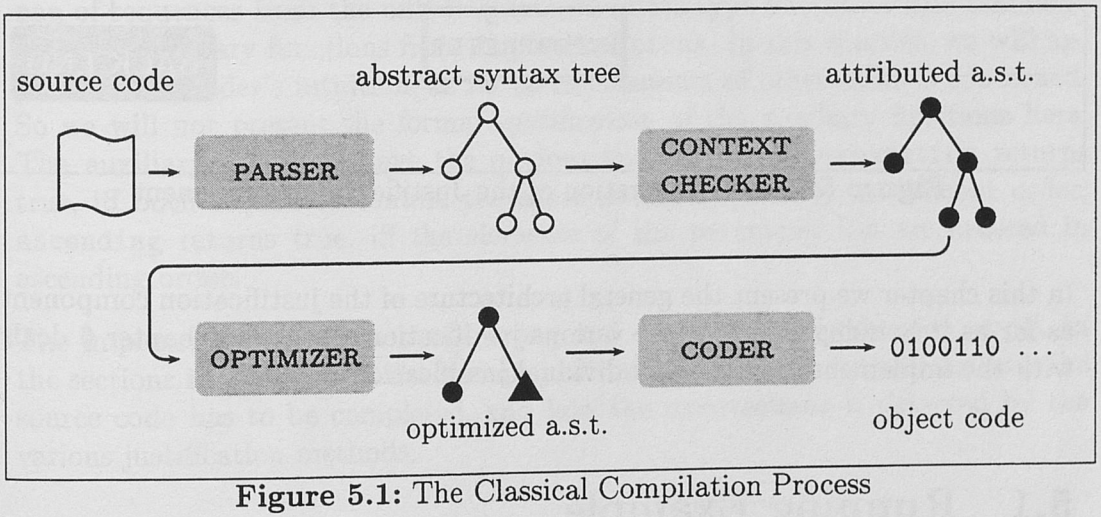

Figure 5.1: The Classical Compilation Process

This picture omits some details – the parser is often split into a scanning and a parsing phase, optimizations are also performed on object-code level, and the coder will probably use other intermediate representations to translate the abstract syntax into object code. But it serves to give an idea of the abstract representations available during the compilation process. It will be easier to integrate a justification method, if we do not have to invent a new intermediate language.

The compilation process can be roughly divided into an analysis and a synthesis phase. During the analysis the compiler checks (analyzes) whether the input actually constitutes a valid program of the respective programming language. During the synthesis phase the result of the analysis phase is processed and transformed into the target language.

The analysis is done incrementally. The compiler first checks regular and context-free properties of the input language before the more expensive check for context-sensitive properties is done. The justification component checks even more general properties than can be expressed by a context-sensitive language. Hence, the justification component fits naturally in the analysis phase right after the context checker.

As a first approximation we conclude that ***the justification component is best situated between the context checker and the optimizer.***

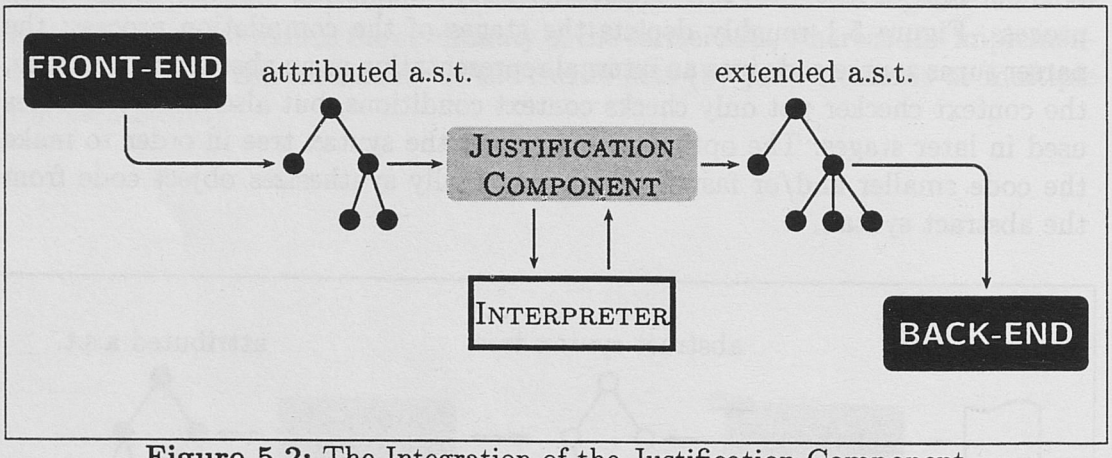

Figure 5.2: The Integration of the Justification Component

In this chapter we present the general architecture of the justification component as far as it is independent of the various justification methods. Chapter 6 deals with the implementation of the individual justification methods.

5.1 Running Example

We will use the sort function as a running example throughout this and the following chapter for the illustration of our concept. Program 5.1 shows the source code for this example. Line numbers are included for later reference. Chapter 7 contains additional examples, in particular, the set instantiation and the deque implementation from the introduction.

Program 5.1 Sorting Lists

```
SIGNATURE SortList                                                    1
  IMPORT Seq[nat]              ONLY seq                               2
         Nat                   ONLY nat                               3
         SeqNatFunctions       ONLY permutation ascending             4

  FUN sort: seq[nat] → seq[nat]                                       5
      SPC  sort(S) = T                                                6
      PRE  true                                                       7
      POST S permutation T AND ascending(T)                           8
-------------------------------------------------------------------------
IMPLEMENTATION SortList                                               1
  IMPORT Seq[nat] ONLY seq ◊ ::                                       2

  DEF sort(◊)    == ◊                                                 3
  DEF sort(a::R) == a::sort(R) !                                      4
```

The sort example consists of two OPAL units, SIGNATURE `SortList` and IMPLEMENTATION `SortList`, the first one being the interface unit, and the other one the corresponding implementation unit. SIGNATURE `SortList` is implemented by IMPLEMENTATION `SortList` as defined in Section 3.3.4.

The interface unit imports some items from other units, namely the data type `seq` of sequences from the unit `Seq[nat]`, the data type `nat` of natural numbers, and two auxiliary functions from `SeqNatFunctions`. In this chapter, we will appeal to the reader's intuition, as far as the contents of other units is concerned. So we will not present the formal specification of the auxiliary functions here. The auxiliary functions have the obvious specifications: `permutation` returns true, iff both sequences contain the same elements, possibly in different order; `ascending` returns true, iff the elements of the parameter list are ordered in ascending order.

The implementation is obviously wrong and contains no justification at all. In the sections that deal with the various justification methods, we will see how the source code has to be completed, and how the incorrectness is detected by the various justification methods.

5.2 Syntax

Before we explain the three phases of the justification component, we need to introduce some syntax for the new entities. The syntax introduced here is also used by the OPAL/J prototype.

Named Algebraic Formulas Algebraic formulas are introduced by the keyword LAW, followed by a name and the formula itself. For example, the fact that `sort` is an idempotent function can be expressed by the following formula:

```
LAW idempotent == ALL S. sort(sort(S)) ≡ S
```

Algebraic formulas are already part of OPAL. OPAL allows also anonymous formulas, but this is forbidden in our framework - we need to be able to address every formula separately.

The syntax for function specifications, however, is new. The signature of `Sort` (Program 5.1) contains a function specification in lines 6-8. This specification is equivalent to the following laws (the premise comes from the given precondition, the conclusion of `Spc[sort]` comes from the postcondition):

```
LAW Dfd[sort] == ALL S. true ⟹ DFD sort(S)
LAW Spc[sort] == ALL S. true ⟹ LET T == sort(S)
                               IN  S permutation T AND ascending(T)
```

The names for these laws are constructed from the name of the associated function. We will use names like these for all formulas that are introduced implicitly, e. g. by a data-type declaration or by the declaration of an algebraic relation.

The declaration by the keyword LAW carries no information whether the formula should be regarded as an axiom or as a theorem. One might consider introducing additional keywords AXM and THM, so that the user can classify the formulas accordingly. If the law `idempotent` is added to the interface of Program 5.1, we could declare that it should be a theorem. This eases the later proof of the implementation relation between the program units in Program 5.1, because we can omit theorems in the implementation proof. On the other hand, if the correctness of the unit is shown by constructing a model, all algebraic formulas are theorems (see Section 3.4.1) and the user has no freedom to decide which formulas are axioms and which are theorems. In this case, the introduction of special keywords merely introduces additional possibilities to make errors.

Proof Declarations Proof declarations are introduced by the keyword PROOF, followed by the name of the proof declaration. As a first example, consider the following proof declaration:

```
PROOF spc_sort : Freetype[seq[nat]] ⟹ Spc[sort]
```

This declares a proof named `spc_sort`, which proves that the formula named `Spc[sort]` can be proven from the premise `Freetype[seq[nat]]`. This latter formula is implicitly declared by the data-type declaration of the free type `seq[nat]`. A proof may employ a set of formulas as premises. The names of these formulas

are separated by spaces as customary in OPAL. A proof declaration may even have no premises at all, in particular if the associated justification is a certification or a formal test.

Justifications Proof declarations must be complemented by a justification, so that the compiler can check the validity of the justification (and is not forced to invent a justification by itself). The justification is introduced with the keyword JUSTF:

```
JUSTF spc_sort == ...
```

The " ... " part, i. e. the justification itself, is dealt with in Chapter 6. The name of the justification must match the name of a proof declaration. We speak of a "justification that is associated with a certain proof declaration".

As a first step towards full automation, we consider the possibility to let the user declare a "default" justification. This default justification is used every time no justification has been explicitly associated with a proof declaration.

Theories For theories we use the same syntax as for signature parts. Only the introducing keyword is changed from SIGNATURE to THEORY.

An assertion is introduced with the keyword ASSERT. For the denotation of the renaming morphism, we (ab)use the standard OPAL syntax for instantiation. Instead of ASSERT `TotalOrder` RENAMED BY $\alpha \to$ `nat` $\lhd \to <$ we write shorter ASSERT `TotalOrder`[`nat`, $<$].

5.3 Phases of the Justification Component

The general architecture of the justification component is given by the three main tasks the justification component has to take care of:

- First, the justification component must *collect* the information that is essential for justifying the correctness of the current unit, but has been neglected by the classical context checker.
- Then, the *unit correctness check* is performed: this concerns the proof declarations given by the programmer.
- Finally, and most importantly, the *justification correctness check* is done: this phase checks whether the justifications in the source code are valid justifications.

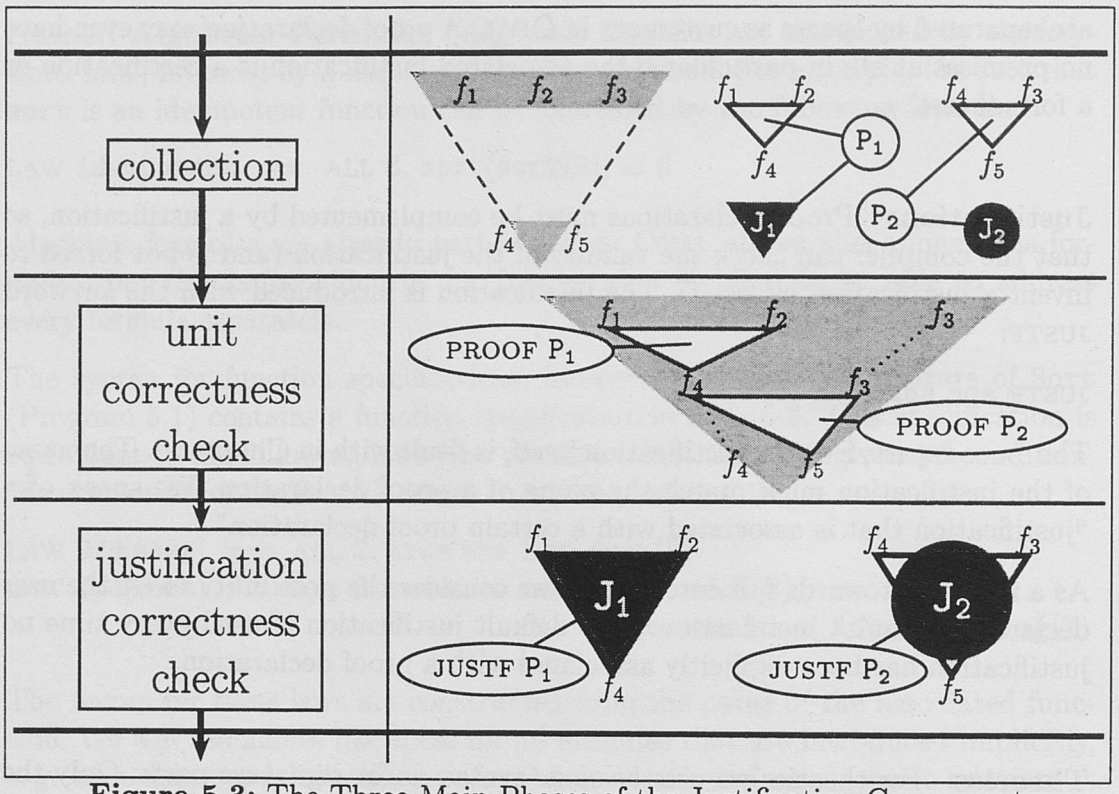

Figure 5.3: The Three Main Phases of the Justification Component

Figure 5.3 shows the three phases together with a graphic illustration of the information checked. In the example we suppose that the source contains the following laws, proof declarations and justifications:

```
LAW f1 == ...                    PROOF P1 : f1 f2 ⟹ f4
LAW f2 == ...                    JUSTF P1 == J1
LAW f3 == ...
LAW f4 == ...                    PROOF P2 : f3 f4 ⟹ f5
LAW f5 == ...                    JUSTF P2 == J2
```

- After the collection phase the compiler knows the named formulas that are visible in the current unit, and has collected the proof declarations and the associated justifications. Most important, the compiler has partitioned the formula set into axioms and proof obligations. In the illustration, the axiom set consists of the formulas $\{f_1, f_2, f_3\}$, the set of proof obligations is $\{f_4, f_5\}$. Those form the top and the bottom of a proof tree. Moreover, the compiler collected the proof declarations P_1 and P_2 with justifications J_1 and J_2 respectively.

- During the unit correctness check, the compiler tries to construct a provisional proof tree from the proof declarations in the source code for the

correctness check $F_{AX} \Longrightarrow F_{TH}$. The shaded triangle represents the proof tree for $F_{AX} \Longrightarrow F_{TH}$, with the axioms at the top and the proof obligations at the bottom. The hollow triangles represent the proof trees for the respective proof declarations, which are in this stage only assumed to exist.

- The justification correctness check ensures that the provisional proof tree constructed is sound. In the illustration of Figure 5.3 justification J_1 is valid, but justification J_2 does not "fit" the proof declaration it is supposed to justify.

5.4 The Collection Phase

The first task is to create the environment necessary for the check of unit and justification correctness. This environment contains the formulas, the proof declarations, and the justifications that are visible in the current unit as well as the information which formulas are axioms and which formulas are proof obligations.

Recall that we divide the formulas into four groups: constructive formulas, algebraic formulas, external formulas and relational formulas. These formulas are partitioned into axioms and proof obligations according to the table in Figure 3.2. If the syntax allows the explicit declaration of axioms and theorems by the user (as opposed to unspecified laws, see Section 5.2), this declaration must be respected as well.

As far as proof declarations and justifications are concerned, things are more straightforward. In principle, proof declarations and justifications are all contained explicitly in the source code. However, sometimes it is useful to add implicit proof declarations and justifications.

In the case of correctness proof by algebraic implementation, the lifting by $\mathcal{F}^{-1}$ in general changes an axiom f of the interface to the proof obligation $\mathcal{F}^{-1}(f)$. However, there are some simple criteria that allow to automatically decide whether $\mathbf{i}(f) \Longrightarrow \mathcal{F}^{-1}(f)$ holds. To make this implication accessible, we add an implicit proof declaration PROOF •`Incl[f]` $\Longrightarrow$ `Lift[f]`. The user can now decide whether to prove $\mathcal{F}^{-1}(f)$ or $\mathbf{i}(f)$. Most often, $\mathbf{i}(f)$ is simpler than $\mathcal{F}^{-1}(f)$, so the user will decide for $\mathbf{i}(f)$, but this is not always possible: even if the implication holds, $\mathbf{i}(f)$ need not be valid. See Section 7.3 for an illustrative example.

The result of the collection phase for our running example is shown in Figures 5.4 and 5.5. Normally this output is not printed during the compilation. User input is printed in *italics*, output of the program is printed in `typewriter` or `underlined typewriter`. We use wavy underline to highlight formulas. In Fig-

ure 5.5, the differences to the environment of the interface are highlighted this way.

The signature is proven by providing an implementation. The formulas Spc[ascending] and Spc[permutation] are imported from SeqNatFunctions. For every imported free type, we have the respective laws. This concerns the explicitly imported sorts nat and seq[nat], and the implicitly imported sort bool. The dots abbreviate some auxiliary laws from SeqNatFunctions.

In the implementation, we have the following changes: The definitional equation for function sort (denoted Def[sort]) is added to the axioms. The specification Spc[sort] has been removed from the axioms. Instead, $\mathcal{F}^{-1}$(Spc[sort]) (denoted Lift[Spc[sort]]) has been added to the proof obligations. As described above the additional proof declaration i(Spc[sort]) $\Longrightarrow$ $\mathcal{F}^{-1}$(Spc[sort]) has been added to the environment. The definedness law Dfd[sort] is treated similarly.

The environment for these toy examples is rather short. This is due to the fact that only very few names are imported. In general the lists of imported items are much longer, and thus the number of laws imported from other units increases quickly.

```
>jcheck-info env SortList.sign
Axioms:
  {Spc[sort],Spc[ascending],Spc[permutation],Freetype[bool],
   Freetype[nat],Freetype[seq[nat]], ... }
Proof obligations:
  {}
Proof declarations:
  {}
Extra proof declarations:
  {}
```

Figure 5.4: The Justification Environment of SortList.sign

5.5 The Unit Correctness Check

This phase checks the correctness as far as this is possible without taking the justifications into account. The information available is the set of formulas, the knowledge which formulas are axioms, and which formulas are proof obligations, and the proof declarations, i. e. the justifications the user has "promised" to deliver. Since the check not only involves a single justification, but checks the correctness of the unit as a whole, we call this phase *unit* correctness check.

```
>jcheck-info env SortList.impl
Axioms:
  {Def[sort],Spc[ascending],Spc[permutation],Freetype[bool],
   Freetype[nat],Freetype[seq[nat]], ... }
Proof obligations:
  {Lift[Spc[sort]],Lift[Dfd[sort]]}
Proof declarations:
  {}
Extra proof declarations:
  {Incl[Spc[sort]] ==> Lift[Spc[sort]],
   Incl[Dfd[sort]] ==> Lift[Dfd[sort]]}
```

Figure 5.5: The Justification Environment of `SortList.impl`

The check consists of building the proof tree for the implication $F_{AX} \Longrightarrow F_{TH}$, with the help of the proof declarations. Since F_{AX}, F_{TH} and the set PD of proof declarations are finite, we can do this by an iterative algorithm.

1. Axioms are trivially derivable laws, so we start with $D_0 = F_{AX}$.
2. In the next iteration we have $D_{i+1} = D_i \cup \{l | \langle l_1\, l_2 \ldots l_n \Longrightarrow l \rangle \in PD \wedge \{l_1, l_2, \ldots, l_n\} \subseteq D_i\}$
3. Stop, if $D_{i+1} = D_i$. We call the final set D.

If $D \supseteq F_{TH}$, the check succeeds.

The algorithm always terminates, because the size of the set D_{i+1} either increases or stays the same. The only formulas that may be added to a set D_i are the target formulas of the proof declarations. The number of proof declarations is finite, and therefore the number of iterations that enlarge the set D_i of the previous iteration is also finite.

If the check fails, we do not only output the set of proof obligations that are not derivable $F_{TH} \setminus D$, but differentiate the reasons. Let *ob* be a non-derivable proof obligation, then we might get one of the following error messages:

- There is no proof declaration with *ob* as the target.
- There is one (or more) proof declarations with *ob* as the target, but at least one of the premises is not derivable either.
- In particular, there may be a (most probably indirect) circular dependency.

Example The correctness check of the signature unit of the sort example (Program 5.1) is performed by providing an implementation. The instantiation has

no proof obligations associated, and therefore all formulas are axioms, and the unit correctness check succeeds trivially.

The implementation, however, is proven correct by model construction. Since there is no proof for the lifted specification and definedness formulas for the function `sort`, the unit correctness check of the implementation of `Sort` (Program 5.1) fails, as shown in Figure 5.6. We see two error messages because the proof obligations `Lift[Spc[sort]]` and `Lift[Dfd[sort]]` are not resolved. The error messages contain additional information. The first error message tell us that there exists a generated proof declaration `Incl[Spc[sort]]` $\Longrightarrow$ `Lift[Spc[sort]]` that could have been used to prove the proof obligation, but it contains a formula `Incl[Spc[sort]]`, which in turn cannot be proven. The second error message contains similar information about the other proof obligation `Incl[Dfd[sort]]`.

```
>jc SortList.impl
ERROR [SortList.impl at unknown location]:
Lift[Spc[sort]] could not be proven:
unproven premises of GeneratedPD[0]: Incl[Spc[sort]]
..no proof declaration with target Incl[Spc[sort]] found
ERROR [SortList.impl at unknown location]:
Lift[Dfd[sort]] could not be proven:
unproven premises of GeneratedPD[1]: Incl[Dfd[sort]]
..no proof declaration with target Incl[Dfd[sort]] found
```

Figure 5.6: The Failing Unit Correctness Check of `SortList.impl`

Chapter 6

Justification Methods

In the previous chapters we have discussed the semantics (Chapters 3 and 4) of correctness checks and the integration into the compilation process (Chapter 5) as far as this is possible without knowing how the proof is performed. To this end, we have introduced proof declarations. By a proof declaration PROOF PD: $l_1\ l_2\ \ldots\ l_n \Longrightarrow l$, the user declares that the validity of l follows from $\{l_1, l_2, \ldots, l_n\}$. The user must complement this declaration by a justification, i. e. a description of a "proof".

The important question here is: What is a "proof"? This is by no means an objective notion. Since the first half of the 20th century logics are classified as intuitionistic or classical because intuitionists do not believe in the law of the excluded middle. In the second half new techniques were introduced into mathematical proofs that stirred up discussions what constitutes a valid proof. Most famous is the proof of the Four-Colour Theorem [AH77, Tho98], which heavily relies on computer support.

The distrust against long and non-human-readable verifications is the central theme in [MLP79], where the authors argue that verifications will never be useful. Mathematical proofs being simple and understandable, they can be "subjected to the social mechanism of the mathematical community"[1]. This polemic paper and the discussion in the following issues of CACM show that there is no universally accepted proof method.

So it is no surprise that there are several methods in use for justifying a program's correctness with respect to its specifications. People with an engineering background most often prefer testing to establish correctness, whereas people with a background in theoretical computer science choose formal proofs. In the be-

[1] Interestingly, the paper does not comment on the – then recent – proof of the Four-Colour Theorem. But the authors do mention the first failed attempt to prove it.

ginning, it was felt that both methods complement each other [GG75], but both communities diverged later on [Sha97].

It is not the purpose of this paper to discuss the merits of different justification methods and choose one or two of them. Instead, we recognize that people from different communities prefer different methods for justifying correctness. Hence, we discuss in this chapter the different methods, with particular focus on the *properties that are important for the implementation*, namely what kind of information must be available to perform the justification.

6.1 The Justification Correctness Check

The final stage of the justification component checks the justifications individually. It is therefore that we call this phase the *justification* correctness check.

Depending on the type of justification, the compiler must be extended with new capabilities. There are the following possibilities:

- Use an external tool.
  - Use a third-party tool. This allows the integration of external knowledge, and saves the implementation of the tool itself.
  - Use a specialized tool. We can tailor the tool to our specific needs, but we have to implement the tool ourselves.

  Since we will probably call the tool several times, it is more efficient, if the external tool can act as a server, answering justification requests from the compiler, which plays the role of a client.
- Integrate the new capabilities into the compiler. This promises to be even more efficient than using a specialized external tool. However, this approach is less flexible than the use of an external tool.

We will discuss the question whether to use a third-party tool or to implement a specialized tool for the integration of support for formal proofs in Section 6.4.4.

6.2 Certification

Certification is "proof by authority" in its purest form. No attempt is made to demonstrate the correctness of the property by its semantics. Instead, the skeptical reader is pointed to an authority that is assumed to carry enough weight

to convince the reader. Whether this is actually the case, users must decide for themselves.

This is actually the biggest disadvantage of certification as a justification method. Correctness of a program is no longer an absolute property of the program, correctness by certification is instead a *user-dependent property.* The authority given in a certification may be accepted by one user and rejected by another, and opinions about the credibility of authorities may change over time. This raises problems if the source code is distributed. Some users might even feel compelled to accept a certificate, although they do not really believe the authority.

The main advantage of certification as a justification method is that it can cope with non-formalized or semi-formalized properties. Hence, certification can be used in the early stages of software development, and later be replaced with other justification methods.

Depending on the kind of authority used we can distinguish different forms of certification.

6.2.1 Referring to a Common Authority

The certification itself points to a common authority, e. g. a mathematical textbook, where the real proof can be found. This allows to embed mathematical theorems that have been proven, even if the proof is too long, too complicated, or simply relies on yet unformalized notions.

As example we take the "Chinese postman problem", which calls for finding a shortest non-simple cycle in a graph that contains every edge at least once. The following lemma [vL90] describes an algorithm to solve this problem:

> *The Chinese postman problem can be solved by means of an all-pairs shortest path computation, solving a minimum weight perfect matching problem, and tracing an Eulerian cycle in an (Eulerian) Graph.*

The correctness proof of this algorithm is rather long:

> In analogy to Theorem 2.12(ii) the minimum cost set of edges that must be doubled in G to make G "Chinese postman optimal" must be the union of minimum cost paths connecting nodes of odd degree, and (conversely) any union of this kind added to G will yield an Eulerian graph. Thus, let A be the set of nodes of odd degree and solve the (A, A)-shortest path problem in G. To select a minimum cost set of paths, design a complete graph G' on node set A with $w'(i, j)$ equal to $d(i, j)$ for $i, j \in A$ (the shortest distance between i and j in G), and determine a minimum weight perfect matching in G'. (Note that, necessarily, $|A|$ is even.) For every edge (i, j) of the matching, double the edges of the shortest path from i to j in G. Tracing an Eulerian cycle in the resulting graph will yield an (optimal) postman walk.

If the algorithm is part of a larger library of graph algorithms, it may be feasible to formalize this proof. But if the algorithm is used to solve an application problem that has no relation to graphs, Eulerian cycles, etc., the situation is different. The choice is either to insert the necessary theory and perform the formal proof, or to insert a pointer to the place where the proof has been done.

The use of certification as a justification method is justified in this situation, because this is the only method that can cope with non-formalized proofs. We can also expect that most readers will accept textbooks as credible authorities.

6.2.2 Taking Responsibility

In the previous section we discussed the situation where a proof does exist, but is not formalized and thus not available to the compiler. Often not even a rigorous proof exists, because the property to be justified is itself not formalized, because only an informal reasoning has been performed, or even because a formal proof might convey too much information about a proprietary algorithm.

In these cases the only other possibility to convince the user of the correctness is the authoritative statement that the property in question actually *is* indeed correct. The only thing left to convince a skeptical reader is knowledge of the person who claims the correctness. Knowing the person can help because the reader may trust into the skill of the certifying person, or because a mistakenly certified property has unpleasant (legal) consequences for the certifier.

6.2.3 Implementation Aspects

Obviously it is very important that the certifier is identified correctly, and that the property is indeed the very same property that the certifier declared correct. In order to solve this problem we can adapt methods developed for digital signatures. Roughly speaking, a digital signature is a hash value of the document, which is encrypted by the owner's private key. Users who want to check the validity of the signature use the owner's public key to check the signature [Riv90]. The implementation of certification as a justification method seems straightforward, but there are some problems that make the implementation more difficult than anticipated.

The main difficulty is the necessity to coordinate the check of the digital signature – which is part of the compilation process – with the tool that adds the digital signature – preferably as a part of an integrated development environment (*IDE*). In the previous section we sketched the general idea for certification:

- First, the proof declaration to be justified is turned into a textual representation.
- Then, this document is digitally signed by the author (or another person willing to take the responsibility).
- The digital signature is inserted into the source code.

The validation of the certification is performed in the following steps:

- The proof declaration to be justified is turned into a textual representation.
- The digital signature is checked with respect to the textual representation.
- If the signature is good, the author is checked against the list of accepted authors.

The problem is the generation of the "textual representation". There are two independent tools that must be able to generate identical representation from different views on the program. The certification tool has access to the source code, together with all layout information, like white space characters, and with comments, documentation, etc. The compiler on the other hand uses a more or less abstract view on the source code.

It would be an advantage, if we could insert the certification component earlier into the compilation process where easier access to the source code is possible. Still, we would have to adapt the parser of the compiler, because comments and white space are usually skipped. But this is not enough: the compiler can only decide which part of the source code is a certification after the parser has finished. Knowing which part of the source code is a certification does not suffice, we have to find the corresponding proof declaration. This search is done with the help of the names of the proof declaration and the justification (which must match), and can only be carried out if the abstract syntax tree is available.

The implementation of the certification method must take this problem into account. The compiler and the certification component must be able to construct the same textual representation of the goal for certification and validation respectively.

A further difficulty arises, because the target of the proof declaration typically depends on items (types, functions, laws) that are declared in other units. The certification is only valid, if none of those items has changed since the certificate was issued. Hence, support for certifications requires that the compiler keeps time stamps for every item in every unit. Usually changes to the source code are checked on the unit level only. For our purpose this is too coarse. Everyone has had the experience that a change to a comment can cause the recompilation of many other units. Since the recompilation is automatic (and hardware is getting

faster), this is acceptable. Certification, however, is not automatic, so any change to an imported (or otherwise used) unit, has the consequence that the user must check and confirm every certification.

Finally, the compiler must know about the user's opinion, which authors, or which authorities are to be trusted and which are not. The compiler must have access to a user-specific data base, which contains the trusted authorities[2].

In the simplest case, the data base contains the names of the authorities accepted. This information may be augmented, for example with an expiration date, or with a restriction for certain units or subsystems (groups of units).

The "verification" of the certificates can be started as soon as the compiler knows which certificate belongs to which proof declaration. Figure 6.1 shows how we could add the certification checker to the compiler as depicted in Figure 5.1.

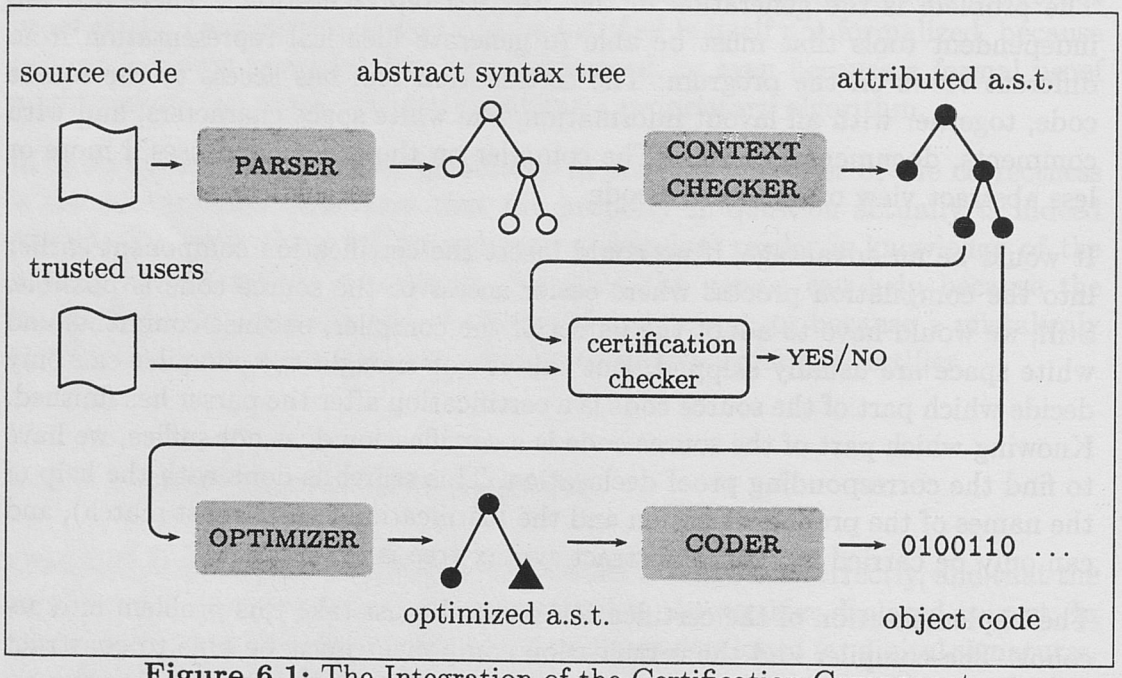

Figure 6.1: The Integration of the Certification Component

It is our aim to base the justification component on the attributed abstract syntax tree that is available after the context checker has successfully completed. It is no problem to check the authors with respect to the user's certification data base with the information contained in the abstract syntax tree. *But we must take care to carry enough information in the abstract syntax tree to reconstruct the textual representation of the proof declarations that are to be justified by certification.*

[2]Note that a compiler cannot ask a user interactively about the acceptability of a digital signature, like web browsers can do.

6.2.4 The Certification Component

This check of a digital signature is best performed by an external tool for the following two reasons: Input and output consist of simple text that requires no expensive parsing, and second, an external tool can also be used by the integrated development environment for the generation of the digital signature.

The check of a certification is divided into three parts: extraction, call and evaluation, see Figure 6.2.

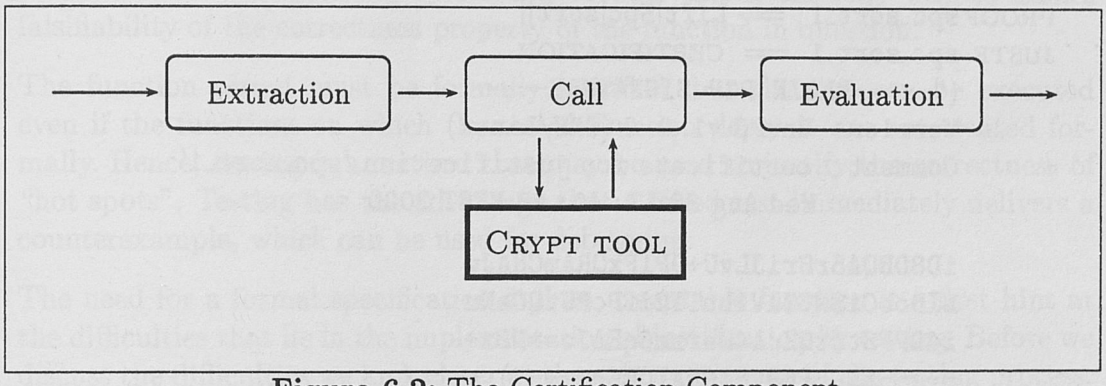

Figure 6.2: The Certification Component

- The *extraction* phase makes the source code of the associated proof declaration and the digital signature available to the external tool. As described above, the compiler must be able to reconstruct the source code from the abstract syntax tree.
- The external tool is *called* and checks the digital signature with respect to the source code of the proof declaration.
- The result of the tool is *evaluated* for the answers to the following questions:
  - Is the signature valid at all?
  - Is the author of the signature among the trusted authors?
  - Is the certification newer than all items that it references?

As discussed in the previous section, the compiler will often lack the means to answer the last question.

Example The justification of the `sort` function by certification is shown in Program 6.1[3]. OPAL/J uses the GNU Privacy Guard (gpg, see [FSF00]) as an external tool. For compatibility reasons the signature is nevertheless dubbed a

[3]Some line breaks have been added for better readability.

"PGP" signature. Since the certification is rather long, the editor should be able to hide the certification.

Program 6.1 Sorting Lists - Certified

```
IMPLEMENTATION SortList
  IMPORT Seq[nat] ONLY seq◊ ::

  DEF sort(◊) == ◊
  DEF sort(a::R) == a::sort(R) ⁉

  PROOF spc_sort_1 : ⟹ Lift[Spc[sort]]
  JUSTF spc_sort_1 == CERTIFICATION
        ("-----BEGIN PGP SIGNATURE-----
        Version: GnuPG v1. 0. 0 (GNU/Linux)
        Comment: certificate for justification 'spc_sort_1'
                 Wed Aug 30 17: 40: 18 MEST 2000

        iD8DBQA5rSriJLvU+9PlFxQRAmCHAJw
        NIDobCiSM63lV1huTT2ifPcPUtQCeNu
        i42PFStStQZuA4marzm6qEAJo==N8z+
        -----END PGP SIGNATURE-----")
```

The result of checking the justification `spc_sort_1` is shown in Figure 6.3. The signature is "good" but `"Klaus Didrich (TU Berlin) <kd@cs.tu-berlin.de>"` is not registered as a trusted certifier. Therefore an appropriate error message is generated.

```
>jc SortList.impl
  checking certification of spc_sort_1
    gpg: Signature made Wed Aug 30 17:40:18 2000 MEST using DSA key ID D3E5171
    gpg: Good signature from "Klaus Didrich (TU Berlin)
    <kd@cs.tu-berlin.de>"
ERROR [SortList.impl at 17.7-17.16]:
author 'Klaus Didrich (TU Berlin) <kd@cs.tu-berlin.de>' not registered
```

Figure 6.3: The Output of the Certification Check

6.3 Testing

Another possibility to establish trust in the correctness of an implementation is collecting *empirical evidence* that the function performs as expected. Put into other words, the implementation is tested with various input values, and the output is checked afterwards whether it meets the specification. The more input values are used and the more "difficult" these input values are, the higher is the

trust in the correctness of the implementation. (We will restate this sentence more precisely below.)

Correctness is thus seen as an empirical science, not unlike a scientific theory. Following Popper ("I shall certainly admit a system as empirical or scientific only if it is capable of being tested by experience." [Pop72]), the correctness cannot be *proven*, the correctness can only be "corroborated" by (successful) past experience. Popper concludes that *falsifiability* is the crucial property of an empirical system. We therefore demand that a function must be specified, if its correctness is justified by testing, because this is the only way to ensure falsifiability of the correctness property of the function in question.

The function tested must be formally specified, but the test can be executed even if the functions on which the tested function depends are not treated formally. Hence, testing is a method that may be used to justify the correctness of "hot spots". Testing has the advantage that a failed test immediately delivers a counterexample, which can be used for debugging.

The need for a formal specification and an executable function is a first hint at the difficulties that lie in the implementation of justification by testing. Before we discuss the difficulties we look closer at the subcomponents of the testing process.

6.3.1 Test-Case Selection

It is in general impossible to execute a program for all possible input data, because the set of possible input data is very large, possibly even infinite. Hence, a subset of test data has to be chosen, which is not too large, but which nevertheless allows to gain enough empirical evidence for a successful justification. For easier management this task is performed in two stages. First, the *test cases* are determined, then the *test data* is chosen.

A test case is a subset of the input data; the notion is also used for the characteristic predicate of the subset. All subsets together must cover all possible input data. Test cases should be "uniform" [Gau95], i. e. the execution with an arbitrary member of the test case will fail or succeed independent of the choice of test data. This property cannot be guaranteed, otherwise we would have a formal proof of correctness.

In general, test cases must be selected by hand, possibly with the help of an interactive tool (such as CTE [Gri95]). But test cases can also be selected on the basis of the specification (black-box tests) or the implementation (white-box tests). There are several possible criteria for each of the two families of test-case selection, such as statement, branch or path coverage [How87] for white box tests and CDNF (canonical disjunctive normal form) [SC96] for specification-based

testing. [Wei97] discusses several methods for obtaining test cases from Z-based specifications.

6.3.2 Test-Data Selection

After the test cases have been defined, representatives must be selected. In general, the selection is performed by the user, requiring additional checking whether the user-supplied test data cover each of the previously computed test cases.

Another possibility is to *derive* the test data from the test cases. This can be done e. g. by Prolog-like systems or as a by-product of checking consistency of a test case: Let P_{tc} be the test case (i. e. the characteristic predicate of the test case), then $\exists x \bullet P_{tc}(x)$ must be proven before automatic generation of test data can begin. If the proof is constructive, it contains a witness for the existentially quantified variable, and we can use this witness as a test data item for the respective test case.

The automatic generation of test data is expensive and has the disadvantage that it requires the use of further sophisticated tools. This does not fit well with the simplicity of testing as a justification method (as compared, e. g., to program synthesis), and might be rejected by some users for this very reason. In the environment we propose, the derivation of test data from test cases could be integrated, though.

6.3.3 Test Execution

Once the test data has been selected, the function to be tested is applied to each of them. Some precautions have to be taken: the function might be undefined for the respective test data, which may manifest either as a run-time error or as non-termination. See Chapter 8 for a discussion of security issues.

If we are in a pure functional setting, there is no need to establish an external state before the test can be performed, and there is no need to undo changes to the external state afterwards. With imperative languages, the situation is different[4].

6.3.4 Test Evaluation

The test evaluator checks, whether the result meets the specification. For simple specifications that have the general shape ALL x. f(x) ≡ E(x), it is possible to

[4]Some Y2K damage occurred after an otherwise successful test, because the test data corrupted working data bases.

execute the right side of the equation as well, and compare both results automatically.

If the specification has a more complicated structure, we have to prove formally that the result does meet the specification. Still, we can substitute the test data item and the result of the test execution into the specification, and hope that a formal proof of the specialized specification is considerably simpler than a formal proof for the unchanged specification.

6.3.5 Implementation Aspects

A very basic test support is possible on the basis of syntactical transformations alone, but this excludes support for test-case selection and requires an executable predicate for test evaluation.

The situation is different, if we want support for test-case selection and more possibilities for test evaluation. Figure 6.4 shows a first attempt to integrate the testing component into the compiler. The testing component consists of four sub-components:

- The *test-case selector* computes the appropriate test cases. The user may define the test cases directly, but most often the user will denote the type of test to be performed (white-box test, black-box test, etc.) and will leave the computation of the actual test cases to the compiler.
- The *test-data checker* ensures that for every test case there is at least one test input. The test input may be either generated or user-defined.
- The *test executor* calls the function to be tested with every item of the test data and records the results.
- The *test evaluator* checks whether the results meet the specification of the function tested.

The main problem is indicated in Figure 6.4 by the thick arrows. Test-case selection and test evaluation need information from the abstract syntax as well as test evaluation of the result. The execution however depends on the availability of executable code. We can therefore not perform the justification between context checker and optimizer, as proposed in Section 5. Matters are further complicated because the input and output from the execution is represented in binary form, which is different from the abstract syntax.

We have already presented our solution to this problem in Section 2.3. Using an *interpreter* provides a way to carry out the test execution on the basis of the attributed abstract syntax. This solution is feasible, if the interpreter fulfils several conditions:

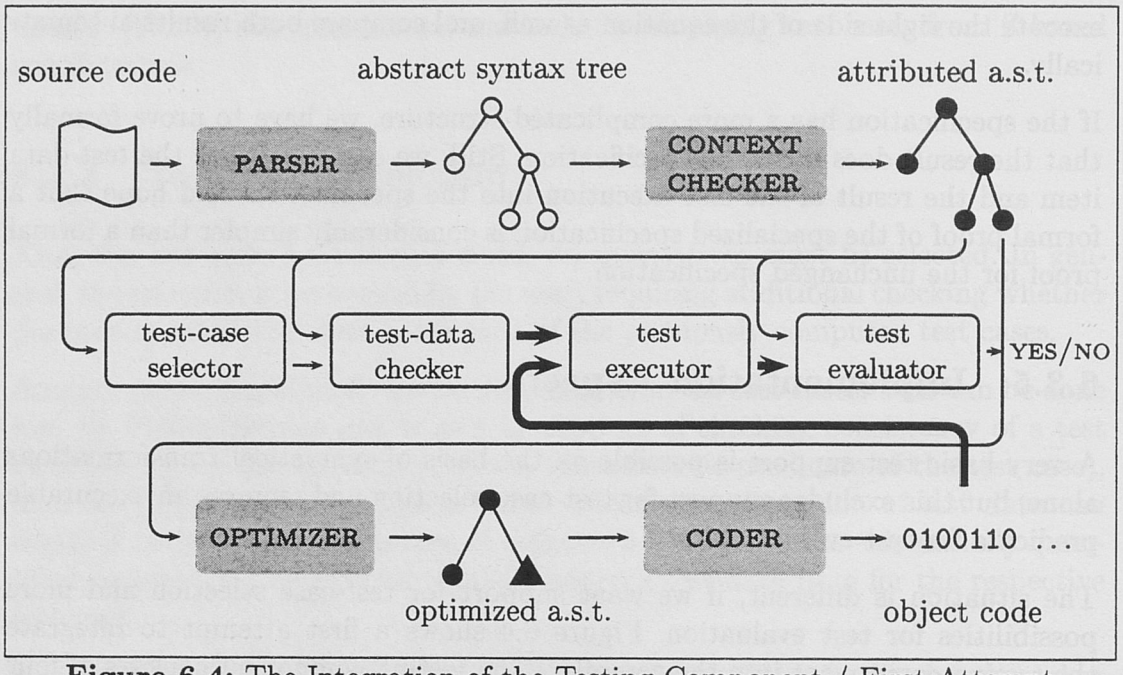

Figure 6.4: The Integration of the Testing Component / First Attempt

- It must be reasonably *fast.* After all, the compilation process is prolonged by several test executions, and if the extra justification effort should be acceptable, the extra time spent on testing should be short.
- The interpreter must be *faithful*, i. e. the execution of the abstract syntax by the interpreter must be semantically equivalent to the code generated by the compiler. This seems to be self-evident, but we must stress here, that "equivalent" includes errors.
- The interpreter must be *secure.* The interpreter must be able to cope with
  - aborting functions
  - non-terminating functions
  - malicious functions

  As long as we stay in a pure functional setting, we need not worry about malicious functions. Pure functions cannot change the external state of a software system, they cannot delete files, or send secret data over the Internet. For the other two cases, we will assume that the interpreter is able to cope with these situations, and postpone a discussion to Chapter 8.

Figure 6.5 shows the improved integration of the testing component into the compilation process.

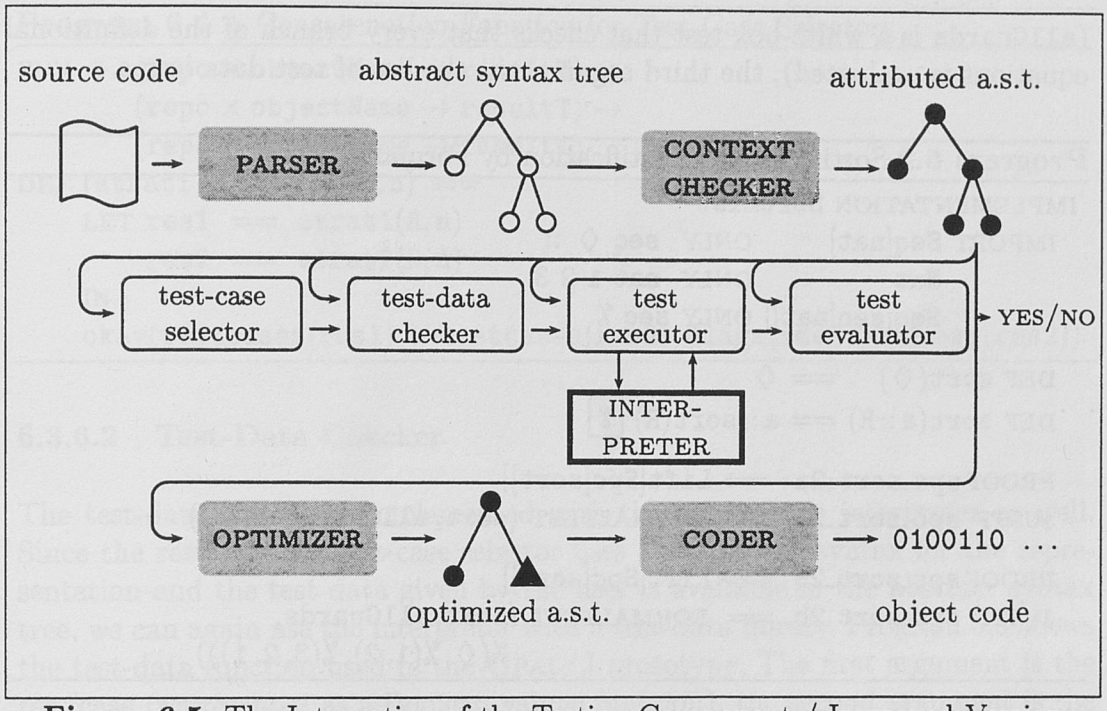

Figure 6.5: The Integration of the Testing Component / Improved Version

6.3.6 The Testing Component

The architecture of the testing component contains the four subcomponents that have been named in the previous section (see Figure 6.6). We have already mentioned that an interpreter will be used as external tool for the test execution. Note that we will use the interpreter as well for the implementation of external tools that implement the test-case selector and test-data checker subcomponents.

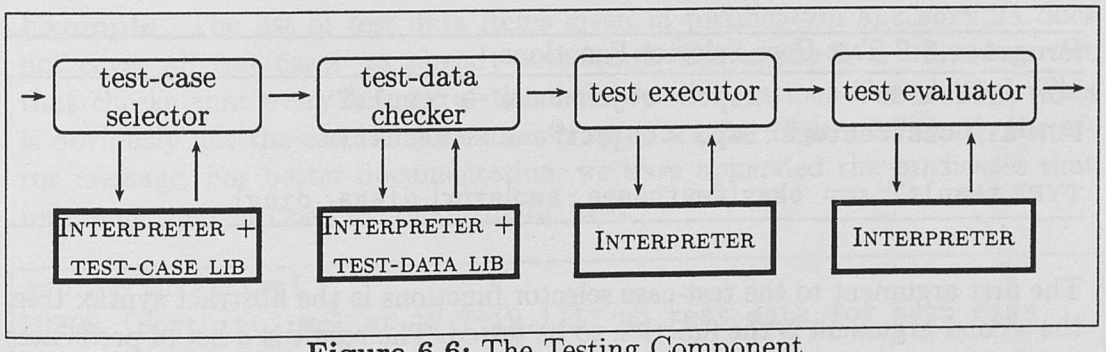

Figure 6.6: The Testing Component

Example Program 6.2 shows how the implementation of `sort` (Program 5.1) can be augmented by a justification for the specification of the `sort` function. The second argument given to FORMALTEST is the heuristic to compute the test cases

(allGuards is a white-box test that checks that every branch of the definitional equations is evaluated), the third argument is the list of test data items.

Program 6.2 Sorting Lists - Justification by Formal Test

```
IMPLEMENTATION SortList
  IMPORT Seq[nat]      ONLY seq ◊ ::
         Nat           ONLY nat 1 2 3
         Seq[seq[nat]] ONLY seq %

  DEF sort(◊)   == ◊
  DEF sort(a::R) == a::sort(R) [!]

  PROOF spc_sort_2a: ⟹ Lift[Spc[sort]]
  JUSTF spc_sort_2a == FORMALTEST (sort, allGuards, %(◊))

  PROOF spc_sort_2b: ⟹ Lift[Spc[sort]]
  JUSTF spc_sort_2b == FORMALTEST (sort, allGuards,
                                   %(◊, %(1,2), %(3,2,1)))
```

6.3.6.1 Test-Case Selection

Test-case selection generally requires intimate knowledge of the abstract syntax of the underlying programming language. We therefore decided to use a specialized tool for the test-case selection. The specialized tool is the interpreter together with a library of functions that compute test cases according to different strategies. The interface of these functions is shown in Program 6.3.

Program 6.3 Test-Case Selector Functions

```
FUN allGuards :        repo × objectName → resultT
FUN allConstructors : repo × objectName → resultT

TYPE resultT == okay(testcases : seq[expr], diags : diag)
```

The first argument to the test-case selector functions is the abstract syntax tree, the second argument is the function to be tested. The result is a list of predicates in an internal representation and, if necessary, error messages.

The use of the interpreter enables the user to extend test-case strategies or to combine them. Of course, the implementation of new strategies requires knowledge of the abstract syntax tree. Program 6.4 presents the definition for a function that concatenates the results of two test-case strategies.

Program 6.4 A Concatenation Function for Test-Case Selectors

```
FUN ⊣⊢ : (repo × objectName → resultT) ×
         (repo × objectName → resultT) →
         (repo × objectName → resultT)
DEF (strat1 ⊣⊢ strat2)(R, n) ==
    LET res1 == strat1(R, n)
        res2 == strat2(R, n)
    IN
    okay(testcases(res1) ⊣⊢ testcases(res2), diags(res1) <+ diags(res2))
```

6.3.6.2 Test-Data Checker

The test-data checker is implemented with the help of the interpreter as well. Since the result of the test-case selector uses the abstract syntax for the representation and the test data given by the user is available in the abstract syntax tree, we can again use the interpreter with a test-data library. Program 6.5 shows the test-data function used in the OPAL/J prototype. The first argument is the test case (represented as a Boolean-valued function), the second argument is the list of test data.

Program 6.5 The Test-Data Checker Function

```
FUN checkTestCases : (α → bool) × seq[α] → bool
DEF checkTestCases(f, ◊) == false
DEF checkTestCases(f, d1::R) ==
      IF f(d1) THEN true ELSE checkTestCases(f, R)FI
```

Example The list of test data items given in justification `spc_sort_2a` does not cover all test cases: as already mentioned, `allGuards` is a white-box test that checks that every branch of the definitional equations is evaluated. This is obviously not the case for justification `spc_sort_2a`. Figure 6.7 shows the error message. For better documentation, we have appended the predicates that represent the test cases for the heuristic used.

```
>jc SortList.impl
ERROR [SortList.impl at 19.7-19.17]: no test data for test case
\\ x_4. ::?[nat](x_4)
>jcheck-info testcases sort allGuards
Test cases for function 'sort' with heuristic 'allGuards'
<\\ x_4. <>?[nat](x_4), \\ x_4. ::?[nat](x_4)>
```

Figure 6.7: An Error Message For a Failed Test

6.3.6.3 Test Execution

For executing the formal test, we take the specification, remove the quantifiers that bind the argument variables to the specified function, replace the arguments given to the specified function with the respective piece of test data and call the interpreter to perform the evaluation.

Example For justification `spc_sort_2b`, the following formulas are given to the interpreter:

```
true ⟹ LET T == sort(◊)          IN S permutation T AND ascending(T)
true ⟹ LET T == sort(1::2::◊)    IN S permutation T AND ascending(T)
true ⟹ LET T == sort(3::2::1::◊) IN S permutation T AND ascending(T)
```

6.3.6.4 Test Evaluator

In general, the result of the test execution is a formula, which must be further simplified. Quite often (and also in our example) the only quantified variables are the function arguments. In this case we can do the whole evaluation automatically.

Figure 6.8 shows the result of checking justification `spc_sort_2b`. The last test case does not fulfil the specification, and we get the appropriate error message.

```
>jc SortList.impl
ERROR [SortList.impl at 22.7-22.17]: test failed for
        input data no. 2 <3,2,1>
```

Figure 6.8: A Failed Test of `sort`

The interpreter must be protected against non-terminating evaluations. Of course, we cannot check termination in advance. A feasible way to prevent non-terminating evaluations is a forced break after a previously defined time-out. This restricts the justification method to those functions that terminate within the defined time-out period.

6.4 Formal Proof

If a formal specification is given, we can formally check whether the implementation meets the specification. The idea of being able to check the correctness formally has two advantages:

- The correctness is not just more or less corroborated, but computed with 100% accuracy.
- A formal algorithm offers more possibilities for automatic support.

This benefit comes at a price: Justification by testing requires a formal semantics for the data-type part of the programming language only. Testing requires a formal semantics for the data so that a formal specification can be given against which the result of the test execution can be checked. A formal proof requires that we are able to reason about the computation encoded in the implementation. Formal proof as a justification method requires a *formal semantics of the* underlying *programming language.*

Theorem-provers are already good at performing lengthy computations of proofs, but for large theories they need assistance in choosing the right assumptions [Gri96, RS97] and in choosing the right derivation at points where no obvious choice exists. Our approach is more explicit and asks the user to provide more information and to present it in a way to the compiler that allows easy addition of the automatically deducible parts.

6.4.1 Representation of Proofs

The representation of proofs poses another problem that must be solved before formal proofs can be used as a justification method in our framework. Currently, formal proofs cannot be computed automatically, therefore we need a notation to include the formal proof (or some parts of it) in the source code.

Formal proofs are most naturally represented as trees with the axioms on top and the goal at the bottom[5]. Figure 6.9 shows such a formal proof tree given as an example in [DF94]. Note that the proof tree has been cut twice in order to narrow the representation. $\mathcal{G}$ is an abbreviation for the definitional equations of the program environment.

The example shows that a proof tree for longer proofs will quickly exceed the available space. The two-dimensional tree representation cannot be integrated easily into the linear representation of source code. Hence, we have to use a linear representation of the proof, which is less human readable, but better suited for automatic treatment.

[NL98a] discusses the efficient representation of proofs in the framework of proof-carrying code. The representation takes advantage of the redundancy in the proof

[5]This is in contrast to other areas of computer science where the leaves are at the bottom and the root is on top.

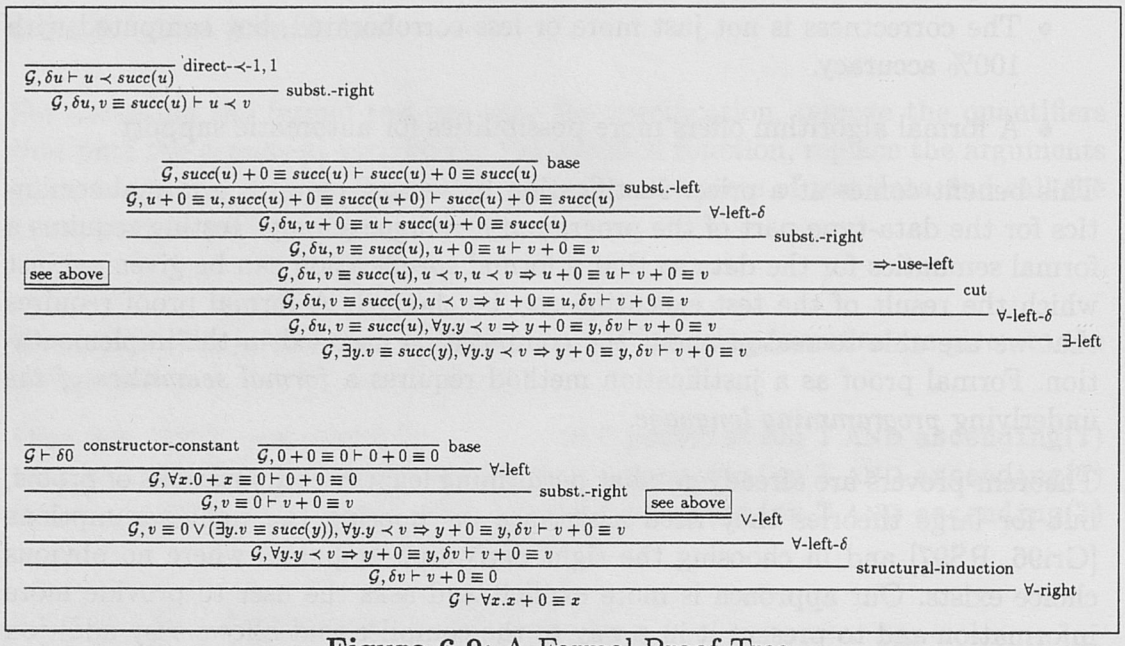

Figure 6.9: A Formal Proof Tree

tree, which is removed in a way that a reconstruction algorithm can easily compute the full representation. Whereas this is a way to deal efficiently with automatically computed proofs, this is no solution for us. The resulting representation is only useful for automatic check, but not for human-guided development.

6.4.2 Tactics and Tacticals

Tactical theorem-provers like ISABELLE system or PVS follow a different approach. These systems use a command language for proof construction. The user combines tactics and tacticals to describe how the proof shall be performed. We will use this way to include proof descriptions in the source code.

The user states the *goal*, i. e. the formula to be proven, and then directs the interactive prover towards the resolution of the goal. The command language consists of *tactics* and *tacticals*. Tactics change the current goal, tacticals combine tactics. Typical tacticals are `THEN` (for sequencing), `ORELSE` (for alternation), `REPEAT` (for iteration).

In addition to the interactive proving facilities, these systems offer the possibility to add new proving tacticals. Tactics are the atomic statements, tacticals the control structures of a *proof engineering language*, which features imperative elements (sequencing, iteration) as well as elements from logic programming (backtracking). The user programs an underlying proof machine in quest of the proof.

The proof engineering languages are quite powerful and the predefined tactics that are delivered with these systems have become more and more sophisticated. Often, only a few tactics must be combined to resolve a proof. The PVS tutorial [COR+95] contains the following statement: "A knowledgeable PVS user can typically get proofs to go through mostly automatically by making a few critical decisions at the start of the proof."

With these powerful tactics, the difference between interactive and automatic proof systems becomes blurred. If only one or two user actions at the beginning of a proof session are required, there is no reason, why the user should not write down these actions and let the system perform its task without interaction.

6.4.3 Implementation Aspects

The integration of the prover into the compilation process is less problematic than the integration of the testing or the certification component. All necessary information is available in the abstract syntax: the goal (the property to be proven) and the proof program given by the programmer.

So we only have to translate the goal and the proof program into the input language of a tactical theorem-prover (like PVS or ISABELLE), call the theorem-prover and await the result.

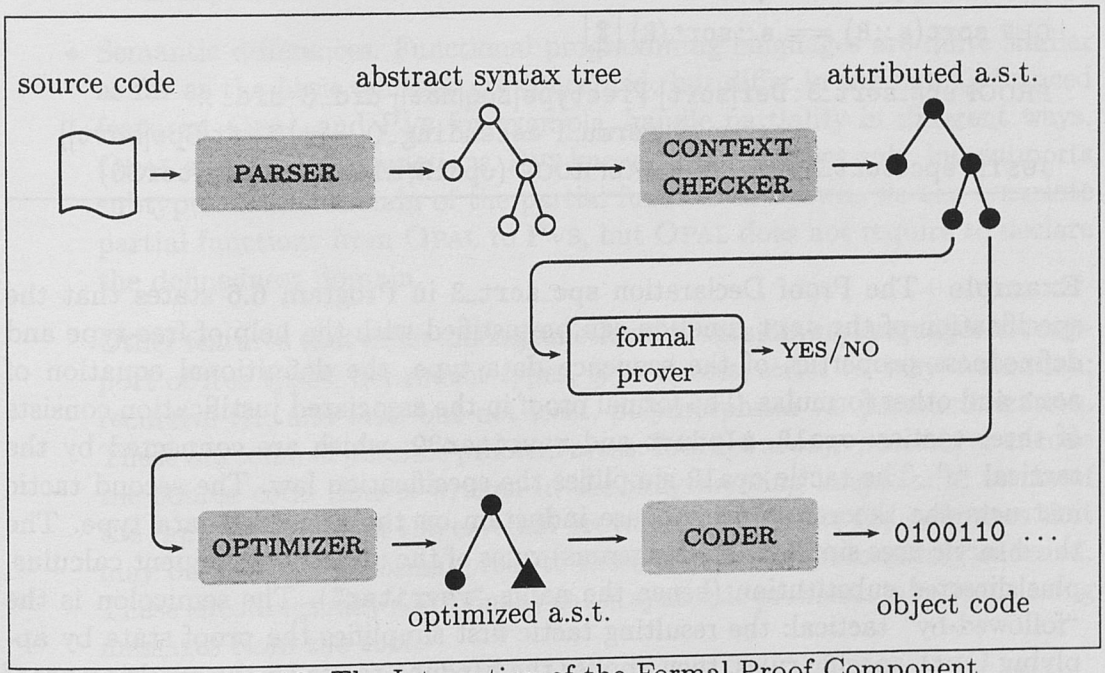

Figure 6.10: The Integration of the Formal Proof Component

6.4.4 The Formal Proof Component

The simplicity of the graphic in Figure 6.11 is not surprising, because the integration of a theorem-prover poses no problems. It might be surprising, though, that the external theorem-prover is not a third-party tool, but that we use again the interpreter with a dedicated library. Before we discuss the pros and cons of our proposal, we present our running example with a formal proof as a justification.

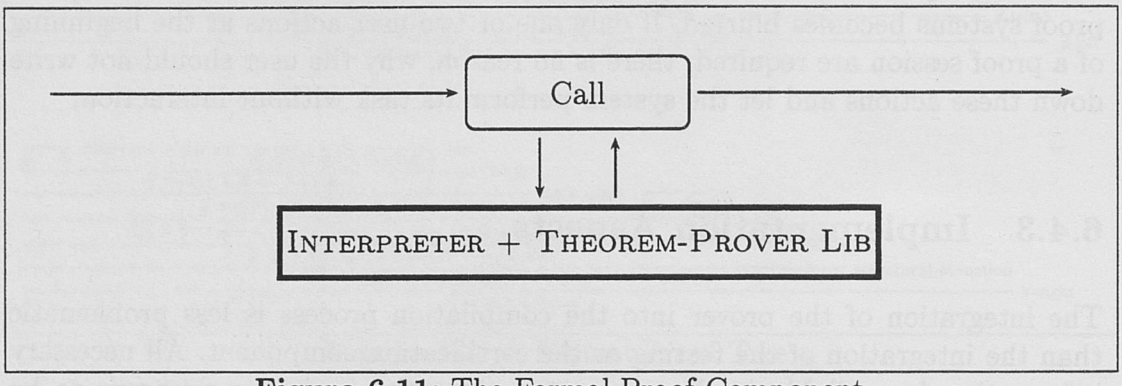

Figure 6.11: The Formal Proof Component

Program 6.6 Sorting Lists - Justified by a Formal Proof

```
IMPLEMENTATION SortList
  IMPORT Seq[nat] ONLY seq◊ ::

  DEF sort(◊)    == ◊
  DEF sort(a::R) == a::sort(R) !

  PROOF spc_sort_3 : Def[sort] Freetype[seq[nat]] dfd_◊ dfd_ ::
                     permu_◊ permu_1 ascending_◊ ⟹ Lift[Spc[sort]]
  JUSTF spc_sort_3 == FORMALPROOF (opalR; mInduct; rewriter30)
```

Example The Proof Declaration `spc_sort_3` in Program 6.6 states that the specification of the `sort` function can be justified with the help of free-type and definedness properties of the sequence data type, the definitional equation of `sort` and other formulas. The formal proof in the associated justification consists of three tactics: `opalR`, `mInduct` and `rewriter30`, which are connected by the tactical ";". The tactic `opalR` simplifies the specification law. The second tactic instructs the theorem-prover to use induction on the sequence data type. The third tactic uses simple (non-dangerous) rules of the underlying sequent calculus, plus directed substitution (hence the name "`rewriter`"). The semicolon is the "followed-by" tactical: the resulting tactic first simplifies the proof state by applying OPAL-specific rules, then applies the `mInduct` tactic on the resulting proof state, and finally calls the third tactic with the result of the induction tactic.

6.4.4.1 Third-Party vs. Specialized Theorem-Prover

Given the number of available theorem-provers, there seems to be no need to develop the $n + 1$-st theorem-prover for our extended compiler. Closer analysis, however, reveals that both third-party and specialized theorem-provers require considerable work on the part of the compiler writer.

The integration of a third-party tool saves the implementation of the theorem-prover as well as the development of a library of proving strategies. But the programming language differs from the input language of the theorem-prover. We must translate the proof declarations and justifications into the input language, and we must be able to do the backwards translation as well: if the formal proof fails, the user needs information about the reason in terms of the programming language.

The following list enumerates some of the difficulties the translation faces:

- Syntactic differences. Even if the abstract syntax hides many details of the programming language, some differences remain, e. g. different rules for identifiers, or different keywords. All these differences can be dealt with by the translation, but often the re-translation is difficult or impossible. This is important if error messages must be re-translated from the context of the third-party tool into the programming language. One aspect of the translation of error messages is the problem of positions, which are most often impossible to translate[6].

- Semantic differences. Functional programming languages are quite similar as far as the basic features are concerned, but differ in the more advanced features. OPAL and PVS for example, handle partiality in different ways. OPAL allows partial functions. PVS knows total functions only, but supports subtypes. If the domain of the partial function is known, we can translate partial functions from OPAL to PVS, but OPAL does not require to declare the definedness domain.

  Other features that differ among functional programming languages are support of free types, dependent types, type classes; strict vs. lazy evaluation; recursive let; and last, but not least, polymorphism vs. parameterization. These are some of possible problem areas. It is not always possible to encode the source code that is written in the programming language in terms of the input language of the external prover. This restricts the programs that may be justified by formal proof to those that can be successfully encoded. These encodings may additionally cause syntactic problems in re-translating messages from the tool.

[6] C has the __LINE__ and __FILE__ preprocessor directives for this purpose.

- Modularization. OPAL uses a two-level modularization structure with an interface unit and an implementation unit, both are connected by an algebraic implementation relation. If this relationship must be encoded in the tool language, this causes again semantic and syntactic translation problems.
- Program library. At first glance, a formal proof of `spc_sort_3` requires only the translation of a few formulas. But these formulas are built over a signature that comprises many units of the programming language. In particular the standard library, but also all of the units of the current project must be translated into the tool's input language.

Some of the translation difficulties can be overcome if a generic theorem-prover (like ISABELLE) is used that allows to define the input language. Still, syntactic and semantic differences remain and make the correctness proof more difficult. [Pus94] contains a study of a verification of a development written in SPECTRUM. With the help of an automated translator, the source code was transformed into an ISABELLE theory, which was the basis for the verification. Some peculiarities of SPECTRUM made the verification quite laborious: SPECTRUM's three-valued logic made the use of ISABELLE's automated prover for predicate logic quite difficult. Some of the automatically generated axioms had to be changed for easier proving.

We checked the derived rules of the OPAL sequent calculus [DF94] with an experimental implementation in ISABELLE (based on the classical sequent calculus LK). For the experiment only a small subset of OPAL was needed; the inclusion of all peculiarities of OPAL would have been more tedious. No precautions were made for a re-translation to OPAL. For the implementation of the basic tactics we had to familiarize ourselves with the ISABELLE environment. Once these preparations were finished we could benefit from the tacticals provided by ISABELLE/LK and could also make use of the possibility of batch proofs.

In Figure 6.12 we compare the cost for the implementation of a third-party theorem-prover, a specialized theorem-prover and the prototype of specialized theorem-prover. Based on our experience we assign a "high", "middle" or "low" cost to each of the parts that must or should be implemented, or "none", if no implementation is necessary.

If we use a third-party theorem-prover, the translation and re-translation has a high cost. The basic tactics for the programming language must be written from scratch, but tacticals and decision procedures for special domains (e. g. natural numbers) come for free, because they are already part of the standard environment. Batch processing is probably not very well supported, the user interface and error handling and debugging facilities assuming that the proof is done interactively.

For a specialized theorem-prover, costs are reversed. The translation and re-translation come for free, whereas all other parts have to be implemented. Note

	third-party theorem-prover	*specialized theorem-prover*	*prototype*
translation	high	none	none
basic tactics	middle	middle	middle
tacticals	none	middle	low
decision procedures	none	high	none
batch proofs	middle	middle	middle
re-translation	high	none	none

Figure 6.12: Comparison of Implementation Costs

that the estimation of a "middle" cost for the implementation of tactics and tacticals depends on the choice for OPAL as implementation language (see the discussion in the following section). For other implementation languages the cost might be "high".

For the prototype of a specialized theorem-prover, we can omit the support of decision procedures and restrict the implementation of tacticals to a few basic tacticals. The cost for complicated proofs is higher, but we have an "early" functional prototype.

There is no clear answer to the question whether to use a third-party theorem-prover or to implement a new one. The costs must be re-evaluated for implementation languages other than OPAL and other languages than functional languages. Especially if the gap between the languages is large or the use of a tactics library or decision procedures is important, it might be more effective to implement two small compilers to translate between the compiled language and the input or output language of the third-party theorem-prover.

6.4.4.2 Comparison between Isabelle and the Opal/J Prototype

Fortunately, we do not have to do the implementation entirely from scratch. If we find a theorem-prover whose implementation language is similar to the language we are processing, we may possibly re-use parts of the implementation.

It is for this reason that we chose ISABELLE as a reference implementation. The implementation language of ISABELLE is ML, a functional language like OPAL. Of course, there are syntactic and semantic differences, but most of them can be expressed within OPAL. The most striking difference – ML's use of polymorphism vs. OPAL's parameterization – can be overcome at the expense of a few additional OPAL units.

More important is the fact that both languages are *strict* languages. ISABELLE does use laziness in its implementation, but since ML is a strict language, this

laziness must be explicitly written down in the source code. So we can re-use this explicit laziness in the re-implementation in OPAL.

We compare three examples taken from the ISABELLE implementation in ML and the corresponding equivalents in OPAL. The first one shows the implementation of a lazy sequence data type, the second presents the functions and data types used for interaction with the user and the compiler and finally we show the implementation of the repetition tactical.

Lazy Sequence Data Type The (central part of the) implementation of the data type is shown in Program 6.7, on the left-hand side the ML implementation of ISABELLE, on the right-hand side the OPAL implementation. The ML implementation is contained in a single file, the OPAL implementation is split into two files, one for the interface and one for the implementation.

Three corresponding pairs of pieces of source code are marked in Program 6.7 to emphasize the similarity between the two implementations. The data-type definition differs in the way both languages denote data-type constructors (postfix in ML – prefix in OPAL), how both languages represent tuples (OPAL needs an auxiliary data type `pair`), and the chosen identifiers (`abs` and `rep` are the usual names for solitary type constructors and selectors in OPAL). The definition of the function `pull` is identical modulo renaming. The definition of `rt` is almost identical modulo renaming; the OPAL implementation is augmented with error handling code.The reader will have no difficulties in finding other correspondences between the two implementations.

Prover Interface The prover interface is compared in Program 6.8. The upper part contains the definition of `tactic` and `proofscript`. The ML definition is shorter for two reasons: OPAL uses additional data types to encapsulate error information[7], and ML uses the `thm` data type, where OPAL uses `seq[sequent]`. The `thm` type is more general than the OPAL counterpart `seq[sequent]`; after all, ISABELLE is a *generic* theorem-prover. For the OPAL implementation, we decided to specialize the data type.

The lower part of Program 6.8 contains the declarations of the top-level function for batch processing. Both declarations need some explanation. The arguments of the ML function `prove_goal` are the context, in which the proof is to be done, the formula (goal) that is to be proven, and a (function that returns a) list of tactics. These tactics are applied sequentially to the goal. The function returns the head of the resulting list of proof states.

[7]The definition of `result` is similar to the definition of the OPAL's I/O monad data type `com`.

Program 6.7 Lazy Sequences

```
signature SEQ =
sig
  type 'a seq
  val make: (unit -> ('a * 'a seq) option) -> 'a seq
  val pull: 'a seq -> ('a * 'a seq) option
  val empty: 'a seq
  val cons: 'a * 'a seq -> 'a seq
  val single: 'a -> 'a seq
  val hd: 'a seq -> 'a
  val tl: 'a seq -> 'a seq
end;

structure Seq: SEQ =
struct
datatype 'a seq = Seq of unit -> ('a * 'a seq) option;

(*the abstraction for making a sequence*)
val make = Seq;

(*return next sequence element as None or Some (x, xq)*)
fun pull (Seq f) = f ();

(*the empty sequence*)
val empty = Seq (fn () => None);

(*prefix an element to the sequence -- use cons (x, xq)
  only if evaluation of xq need not be delayed, otherwise
  use make (fn () => Some (x, xq))*)
fun cons x_xq = make (fn () => Some x_xq);

fun single x = cons (x, empty);

(*head and tail -- beware of calling the sequence
  function twice!!*)
fun hd xq = #1 (the (pull xq))
and tl xq = #2 (the (pull xq));
```

```
SIGNATURE LSeq[α]
  SORT α
        -- % Type declaration
  TYPE lseq == ◇
             ::(ft : α, rt : lseq)
        -- % lazy constructor
  FUN make : (() → option[pair[α, lseq]]) → lseq
        -- % Construct a list
  FUN % : α → lseq
        -- % Accessing Elements
  FUN pull : lseq → option[pair[α, lseq]]

IMPLEMENTATION LSeq
                -- % Type implementation
  DATA lseq == abs(rep : () → option[pair[α, lseq]])

  DEF make == abs
       --%Accessing Elements
  DEF pull(abs(f)) == f()

  DEF ◇ == abs(λ. nil)
  DEF f::R == abs(λ. avail(f & R))
       --%Construct a list
  DEF %(d) == d:: ◇

  DEF ft(s) ==
      IF s◇? THEN ABORT("ft'LSeq : empty sequence")
             ELSE 1st(cont(pull(s))) FI

  DEF rt(s) ==
      IF s◇? THEN ABORT("rt'LSeq : empty sequence")
             ELSE 2nd(cont(pull(s))) FI

  DEF ◇?(s) == nil?(pull(s))
  DEF ::?(s) == avail?(pull(s))
```

The OPAL function `apply` is not meant to be called interactively, hence the many arguments. The first argument corresponds to the first argument of the ML function, `repo` being the name of the data type that represents the abstract syntax tree. The following two arguments serve to construct error messages. The arguments with type `seq[formula]` and `formula` are the premises and the target of the initial sequent. The proofscript encapsulates the tactic to be used. The function applies the tactic to the initial sequent, and checks, whether all subgoals are resolved in the resulting proof state. If not, an appropriate error message is added. The result of the function consists of the error messages only and not of the final proof state itself. Since the compiler and the theorem-prover are different processes, we found it necessary to minimize the information sent between the two processes.

Program 6.8 The Prover Interface

```
(*A tactic maps a proof tree to a sequence of proof trees:
  if length of sequence = 0 then the tactic does not apply;
  if length > 1 then backtracking on the alternatives
  can occur.*)

type tactic = thm -> thm Seq.seq;
```

```
TYPE state == state(...,diags : diag,
                    subgoals : seq[sequent],...)
TYPE result == okay(α : lseq[state])
               fail(lastState : state)

TYPE proofscript == abs(rep : state → result)
```

```
val prove_goal: theory -> string ->
                (thm list -> tactic list) -> thm
```

```
TYPE resultF == okay(..., diags : diag)

FUN apply : repo × unitName × an×
            seq[formula] × formula × proofscript → resultF
```

Repetition Tactical The implementation of the repetition tactical shows both similarities and differences of the implementation languages. Program 6.9 shows the source code of both implementations. Function calls for tracing or debugging have been removed, and some identifiers have been renamed to emphasize the correspondences. It is not possible in OPAL to define local mutually recursive functions. Therefore we have to lift the local definitions and make the tactic argument explicit, which is supplied implicitly to the local functions `rep` and `repq` in ML. Otherwise, the OPAL functions `*`, `repeat` and `repeatq` closely resemble their ML counterparts `REPEAT`, `rep`, and `repq` respectively.

Program 6.9 The Repetition Tactical

```
val REPEAT : tactic -> tactic

fun REPEAT tac =
      fun rep qs st =
        case Seq.pull(tac st) of
            None        => Some(st, Seq.make(fn()=>repq qs))
          | Some(st',q) => rep (q::qs) st'
      and repq [] = None
        | repq(q::qs) = case Seq.pull q of
            None        => repq qs
          | Some(st,q) => rep (q::qs) st
  in  rep []  end;
```

```
FUN * : proofscript → proofscript
DEF *(abs(tac)) == pscript(λst. repeat(◊, tac, st))

FUN repeat : seq[lseq[state]] × (state → result) × state →
             lseq[state]
DEF repeat(qs, tac, st) ==
    IF fail?(tac(st)) THEN %(st) ⊣ (λ. repeatq(qs, f))
    ELSE LET (st1, q) == pull(data(tac(st)))
         IN repeat(q::qs, tac, st1)
    FI

FUN repeatq : seq[lseq[state]] × (state → result) →
              lseq[state]
DEF repeatq(◊, tac) == ◊
DEF repeatq(q::qs, tac) ==
    IF q◊? THEN repeatq(qs, tac)
    ELSE LET (st, q1) == pull(q)
         IN repeat(q1::qs, tac, st)
    FI
```

6.4.4.3 Example

The formal proof of the `sort` function fails. Figure 6.13 shows the result of the justification of Program 6.6. The error message states that the tactical could be successfully applied, but the resulting proof state contains unresolved subgoals. The hint points the human prover to the fact that the resulting lazy sequence of proof states (see Program 6.8) has more than one element. Access to the other elements involves backtracking of the proof tree. Since this is very expensive, it is not done automatically; instead, the user is informed about this possibility.

The error message also lists the unresolved subgoals of the proof. Only the target formulas of the sequents are printed, because the list of premises often is rather long. Full information is contained in a separate file, see Figure 6.14. The OPAL/J prototype provides some additional information about the formulas by colour coding, but the output is still difficult to understand. The goal is expressed in terms of names of formulas (`Def[sort]`, `ascending_◊`, `Spc[sort]`), but the resulting proof state contains many formulas, some of them with generated variable names. An improved OPAL/J compiler should close this gap by provid-

```
>jc SortList.impl
ERROR [SortList.impl at 27.7-27.16]: unfinished proof
  unresolved subgoals are:
  <[ascending((c! :: sort(a!)))], DFD rt(<>), DFD ft(<>),
   EX ft_5_cj:nat rt_4_cj:seq[nat]. (ft_5_cj :: rt_4_cj) === <>,
   EX ft_5_ck:nat rt_4_ck:seq[nat]. <> === (ft_5_ck :: rt_4_ck)>

HINT [SortList.impl at 27.7-27.16]: resulting proof state ambiguous
```

Figure 6.13: An Unsuccessful Proof of Program 6.6

ing higher-level descriptions of the final proof state. This issue is discussed in Section 10.2.4.

```
final proofstate is:
<
[0]
targets: <§0 [ascending((c! :: sort(a!)))], §1 DFD rt(<>),
§2 DFD ft(<>),
§3 EX ft_5_cj:nat rt_4_cj:seq[nat]. (ft_5_cj :: rt_4_cj) === <>,
§4 EX ft_5_ck:nat rt_4_ck:seq[nat]. <> === (ft_5_ck :: rt_4_ck)>

premises: <§0 DFD c!, §1 DFD a!, §2 LAW permu_<>, §3 LAW permu_1,
§4 LAW ascending_<>, §5 LAW dfd_<>, §6 LAW dfd_::,
§7 sort(<>) === <>, §8 ALL ft_5_a0:nat rt_4_a0:seq[nat].
sort((ft_5_a0 :: rt_4_a0)) === (ft_5_a0 :: sort(rt_4_a0)),
§9 Gen[seq[nat] <<>, ::>], §10 Discr[::, ::?], §11 Discr[::, <>?],
§12 Discr[<>, ::?], §13 Discr[<>, <>?], §14 Sel[::, rt],
§15 Sel[::, ft], §16 Equiv[::, ::], §17 Equiv[<>, <>],
§18 DDfd[::?], §19 DDfd[<>?], §20 SDfd[rt], §21 SDfd[ft],
§22 DFD sort(a!), §23 [permutation(a!,sort(a!))],
§24 [ascending(sort(a!))]>>
end of final proofstate
```

Figure 6.14: The Resulting Proof State

The first formula of the proof state exhibits the problem: It is not possible to prove ascending(c::sort(a)). The variables a! and c! are global variables. We chose ! to mark names of global variables, because these are bound by an !! quantifier in ISABELLE. The other formulas are part of the free-type properties of seq[nat]. The variable names reveal that these formulas are automatically generated.

There is a single subgoal [0] with 5 target formulas and 25 premises. Most formulas are those given in the source code: premises §2–§6 are explicitly given,

premises §7 and §8 are derived from `Def[sort]`, premises §9–§21 and targets §1–§4 are derived from `Freetype[seq[nat]]`.

The interesting formulas are underlined: premises §23 and §24 express the induction hypothesis, target §0 is the formula to be proven by induction.

Debugging unsuccessful proofs can be as difficult as debugging errors in function implementations. Even if the immediate reason is clear, it is still difficult to understand the underlying reason. In our small example, this is easy: the equation `sort(a::R) == a::sort(R)` does not fulfil the requirement that the resulting sequence is "ascending".

6.5 Program Synthesis

Program synthesis (or program derivation) is not exactly a method to justify correctness, it is rather a method to *avoid* the need for a separate justification. A justification by a formal proof – as described in the previous section – consists of the property (specification), the implementation and the formal proof. All three parts are supplied by the programmer. The compiler merely checks whether specification, implementation and proof agree.

Program synthesis ensures by construction that the implementation of a function meets the respective specification. This is an advantage and a disadvantage at the same time. It is an advantage, because implementation and justification are done simultaneously, which is more efficient than doing both tasks separately. It is a disadvantage, because we loose redundancy: the need to provide a specification and an implementation and to show that they both agree introduces an additional level of reliability.

Program synthesis is a rather involved technique. The descriptive specification is turned into an implementation. This implies the introduction of control structures (iteration, recursion) that is often the "eureka" step in program development and therefore defies automation. However, an important special case is handled easily. In functional programming, the specification of many auxiliary functions is not only executable but almost identical to the implementation. Figure 6.15 shows an excerpt of the axioms given for the SPECTRUM specification of lists in [BDDG93], namely the axioms for the length function (#) and the concatenation (++).

In these simple cases, there is no need to require that the user writes a separate implementation and a separate justification. This is formal noise. It is more appropriate to skip an explicit implementation and just tell the compiler to re-use the specification.

```
#(<>)          = 0;
#(x ^ s)       = succ(#(s));

<> ++ s        = s;
(x ^ s) ++ t   = x ^ (s ++ t);
```

Figure 6.15: SPECTRUM: The Specification of Some List Functions

The efficiency of synthesized implementations is another point of concern. But experience shows that automatic development need not result in inefficient programs. The AMPHION system [LPPU94] synthesizes programs with "acceptable performance" in a restricted domain area. The KIDS system has been successfully used to develop transportation scheduling algorithms that outperform traditionally developed programs [SP93].

Program synthesis differs from the other three methods in an important aspect. The other methods require that the implementation exists and check whether it meets its specification. It is possible to eliminate this check and run the compiler without the justification component, just as it is possible to compile without optimization for rapid prototyping. If program synthesis is employed as justification technique the situation is different. The existence of the proof is a precondition for the program. We are therefore committed to perform the justification.

6.5.1 Synthesis Techniques

The information available in our application is the specification of the function to be implemented. Hence, we cannot use methods that try to deduct the implementation from examples. This has been studied in the area of artifical intelligence for a long time and is still subject of ongoing research (see, for example, [SW98]).

The general form of a specification for a function f is $\forall x \bullet Pre(x) \Rightarrow Post(x, f(x))$. We can either apply *correctness preserving transformations* to this formula, until we arrive at a formula $\forall x \bullet Pre(x) \Rightarrow (f(x) \equiv I(x))$. If $I(x)$ is executable, we can use I as the implementation for f. A short example, the development of an implementation for the maximum segment sum problem (presented in [Gib94]) is shown in Figure 6.16.

A more general method is *Constructive Programming* [Hue95]. The function f is unskolemized in the specification. This yields the theorem $\forall x \bullet Pre(x) \Rightarrow \exists y \bullet Post(x, y)$. The point is that the proof must be done constructively, i. e. the

$$
\begin{aligned}
mss &= max \circ sum^* \circ segs \\
&= max \circ sum^* \circ flatten \circ tails^* \circ inits \\
&= max \circ flatten \circ sum^{**} \circ tails^* \circ inits \\
&= max \circ max^* \circ sum^{**} \circ tails^* \circ inits \\
&= max \circ (max \circ sum^* \circ tails)^* \circ inits \\
&= max \circ (id, \otimes) \not\rightarrow^* \circ inits \\
&= max \circ (id, \otimes) \not\not\rightarrow
\end{aligned}
$$

Figure 6.16: The Development of Maximum Segment Sum

existence of a y is proven by producing a witness. This witness is used for the implementation.

6.5.2 Implementation Aspects

The integration of a program synthesis tool into the compilation process poses not more difficulties than the integration of the formal proof tool. The main difference is that the result of a successful program synthesis must be inserted into the abstract syntax tree. Since no additional checking is needed for automatically generated implementations we can easily insert the new implementation. Figure 6.17 shows the structure of the compiler that has been extended by a synthesis tool.

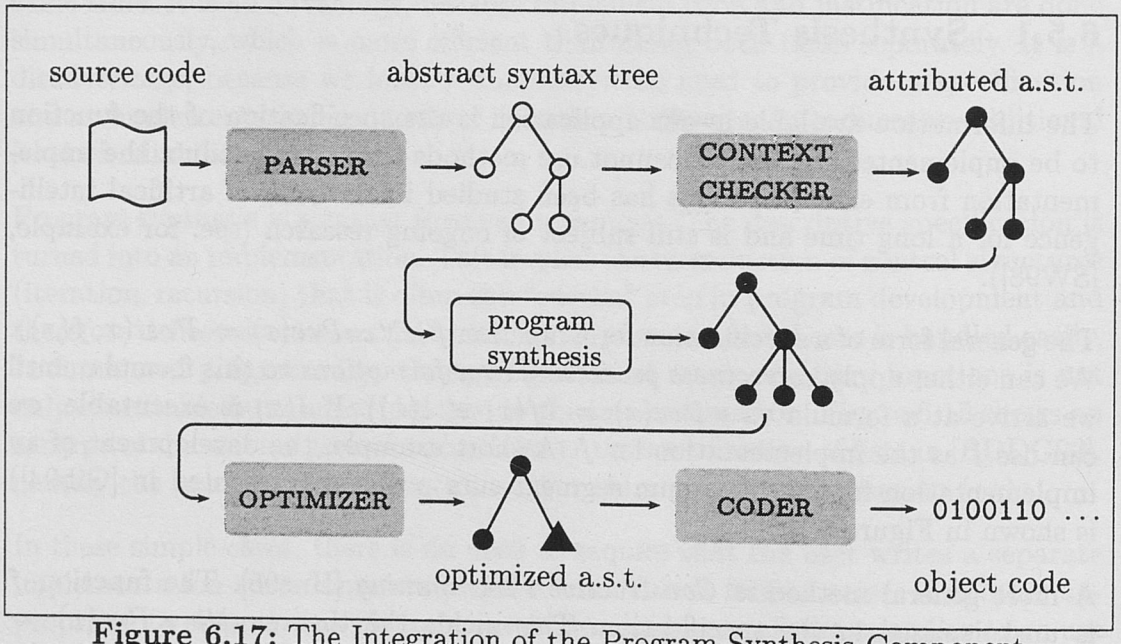

Figure 6.17: The Integration of the Program Synthesis Component

6.5.3 The Program Synthesis Component

The ability to support program synthesis is a by-product of the possibility to perform formal proofs. We support the Constructive Programming approach. Let $\forall x \bullet Pre(x) \Rightarrow Post(x, f(x))$ be the general schema for a function specification. If we find a constructive proof for this formula, we can use the witness for the existence of a y for the implementation.

Figure 6.18 shows the structure of the synthesis component. The resulting function is sent to the compiler as part of the " ... " in the type definition of `resultF` (see Program 6.8).

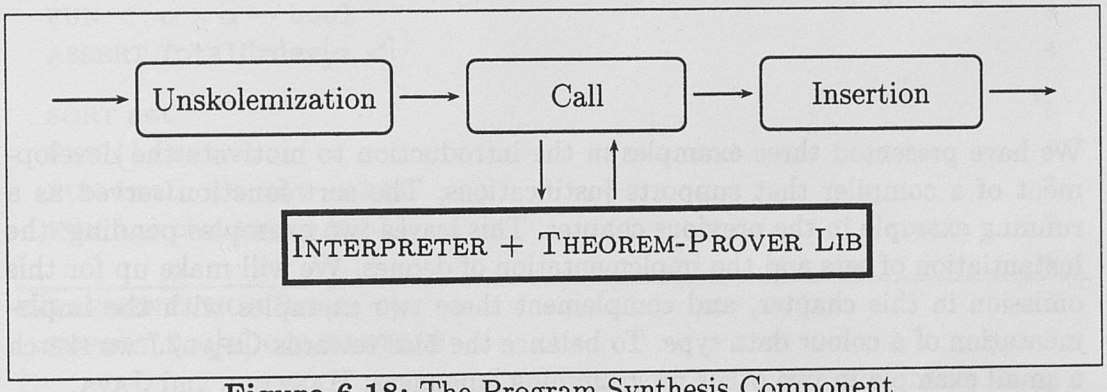

Figure 6.18: The Program Synthesis Component

No Example OPAL/J has only tactics for the synthesis of simple functions (without recursion), therefore we cannot present an example attempt to derive an implementation for the `sort` function.

Chapter 7

Examples

We have presented three examples in the introduction to motivate the development of a compiler that supports justifications. The sort function served as a running example in the previous chapter. This leaves two examples pending: the instantiation of sets and the implementation of deques. We will make up for this omission in this chapter, and complement these two examples with the implementation of a colour data type. To balance the bias towards OPAL/J we sketch a small example in two other programming languages, HASKELL and JAVA.

A longer OPAL/J example, the proof of an implementation of the Sieve of Eratosthenes, is contained in [Did99].

The emphasis in this chapter is not on the justifications. Instead, we want to show how a correctness-aware compiler can assist in developing correct programs. We expect that such a compiler detects the properties that are not fulfilled, and we will see that the failed justifications also point the skilled engineer to the deeper reasons for the failures.

7.1 Instantiation of Sets

This example shows how a compiler can detect errors in the instantiation of data types, and thus prevent the user from committing errors that are difficult to find. Before we can present the example, we want to define the environment of the example, i. e. the OPAL implementation of the set data type.

Part of the implementation of sets is shown in Program 7.1. The signature declares in Line 4 that the parameter must obey the total ordering properties as defined in the theory of total orderings (see Program 7.2).

The implementation uses ordered sequences for the representation of sets. Line 3 defines the data type of sets with the constructor `abs` (short for "abstraction") and the selector `repr` (for "representation"). Line 4 declares that the visible subset of the data type (see Section 3.3.6) is characterized by the predicate `ordered`. This entails proof obligations for the functions that are declared in the interface. We will return to the subject of data-type implementation with junk elements in Section 7.3.

Program 7.1 The Set Data Type

```
SIGNATURE Set[α, <]                                                    1
  SORT α                                                               2
  FUN <: α × α → bool                                                  3
  ASSERT TotalOrder[α, <]                                              4

  SORT set                                                             5
  FUN {}: set                                                          6
  FUN incl: α × set → set                                              7
  FUN in: α × set → bool                                               8
  ...                                                                  9
-------------------------------------------------------------------------
IMPLEMENTATION Set[α, <]                                               1
  IMPORT Seq[α] COMPLETELY                                             2

  DATA set == abs(repr: seq[α])                                        3
  VISIBLE set: ordered                                                 4

  FUN ordered: set → bool                                              5
  DEF ordered(abs(◊))          == true                                 6
  DEF ordered(abs(x:: ◊))      == true                                 7
  DEF ordered(abs(x::y::R)) == x < y  and  ordered(y::R)               8

  DEF {} == abs(◊)                                                     9
  DEF incl(x, abs(◊))    == abs(x:: ◊)                                 10
  DEF incl(x, abs(a::R)) == ...                                        11
  DEF x in abs(◊)    == false                                          12
  DEF x in abs(a::R) == IF x < a THEN false                            13
                        IF a < x THEN x in abs(R)                      14
                                 ELSE true                             15
                        FI                                             16
  ...                                                                  17
```

The implementation of `incl` is omitted for lack of space, but the implementation of `in` is given in Lines 12-16. The implementation exploits the fact that $<$ is required to be a total order and that the representation of sets is done by ordered sequences. The total-order property guarantees that the guards in Lines 13 and 14

Program 7.2 The Theory of Total Orderings

```
THEORY TotalOrder[α, < : α × α → bool]
  SORT α
  FUN < : α × α → bool

  LAW dfd          == ALL x y. DFD x < y
  LAW total        == ALL x y. x < y OR y < x OR x ≡ y
  LAW transitive   == ALL x y z. x < y AND y < z ⟹ x < z
  LAW irreflexive  == ALL x. NOT x < x
```

are defined and that a and x are equal, if the elements cannot be ordered by $<$. The branch in Line 13 skips the rest of the set and returns false immediately, because we know that the searched element cannot occur in the rest of the ordered sequence.

7.1.1 Wrong Instantiation - First Attempt

For the example application we use the colour data type together with the ordering on colours as shown in Program 7.3. We assume that the colour data type is implemented with three constants `red`, `green` and `blue`, as shown later in Program 7.11.

Program 7.3 An Ordering on Colours

```
SIGNATURE ColourOrd
  IMPORT Colour ONLY colour

  FUN < : colour × colour → bool
```

```
IMPLEMENTATION ColourOrd
  IMPORT Colour ONLY colour red green blue

  DEF red < x      == true
  DEF green < red  == false
  DEF green < x    == true
  DEF blue < x     == false
```

An example instantiation is shown in Program 7.4. The instantiation of `Set` with [`colour`, $<$] entails several proof obligations, because the `Set` interface contains the requirement that the parameter meets the total-order properties.

The justification environment of `ColourOrdAppl` (see Figure 7.1) contains several axioms that are imported from `Colour` and the ubiquitous `BOOL`, as well as the proof obligations that are induced by the requirement ASSERT `TotalOrder`[α, $<$]

Program 7.4 Wrong Instantiation of Sets

```
SIGNATURE ColourOrdAppl
  IMPORT Colour          ONLY colour
         ColourOrd       ONLY <
         Set[colour, <]  COMPLETELY ?
  ...
```

```
ColourSetAppl.sign>jcheck-info env
Axioms:
{LAW inv_inv, LAW inv_fix, LAW in_incl[colour, <],
  LAW inv_fix_green, Freetype[bool], Freetype[colour]}
Proof obligations:
{LAW dfd[colour, <], LAW total[colour, <],
  LAW transitive[colour, <], LAW irreflexive[colour, <]}
Proof declarations:
{}
Extra proof declarations:
{}
```

Figure 7.1: The Justification Environment of `ColourOrdAppl.sign`

in SIGNATURE `Set`. Since these proof obligations are not resolved, the compiler issues error messages.

7.1.2 Justifying Total Order Properties

The only way to fulfil these proof obligations is to add an assertion of the total-order properties for the < function from `ColourOrd`. The best place to add the assertion to is the interface of `ColourOrd`. The implementation must be augmented with proof declarations and justifications for these assertions. Program 7.5 shows the resulting source code.

Since the program of `ColourOrd` claims that the function < is a total order, the instantiation in unit `ColourOrdAppl` is now (relative) correct. Of course, the program as a whole is not globally correct. The problem shows up in the justification of the implementation of `ColourOrd`.

The justifications are all done by formal proof. The tactic `mInduct` performs an induction proof, which degenerates to a complete case distinction for the `colour` data type. The other tactics apply logical rules (`rewriter30`, `tApartS`) or properties of the OPAL language (`opalR`).

Program 7.5 The Ordering on Colours With Assertion

```
SIGNATURE ColourOrd
  IMPORT Colour ONLY colour
  ASSERT TotalOrder[colour, <]

  FUN < : colour × colour → bool
IMPLEMENTATION ColourOrd

  DEF red < x     == true ?
  DEF green < red == false
  DEF green < x   == true ?
  DEF blue < x    == false

  PROOF dfd : dfd_false Def[<] ⟹ Lift[dfd]
  JUSTF dfd == FORMALPROOF (mInduct; rewriter30; opalR; rewriter30)

  PROOF total : dfd_false Def[<] ⟹ Lift[total]
  JUSTF total == FORMALPROOF (mInduct; rewriter30; opalR;
                              rewriter30)

  PROOF transitive : Def[<] ⟹ Lift[transitive]
  JUSTF transitive == FORMALPROOF (mInduct; tApartS)

  PROOF irreflexive : Def[<] ⟹ Lift[irreflexive]
  JUSTF irreflexive == FORMALPROOF (mInduct; tApartS)
```

The proofs for definedness, totality and transitivity succeed, but the proof for irreflexivity fails. The final proof state (see Figure 7.2) exhibits the problem to the justification engineer. There are two subgoals to be proven. Both have an empty target list, because the initial target contained a negation. We find the initial target as a premise §9 in both subgoals. Premises §0 - §8 originate from the definitional equation of <.

The underlined premises represent the irreflexivity (NOT `x < x`) for `red` and `green`. And indeed, this is the problem with the implementation of the less function on colours: irreflexivity is violated for `red` and `green` and this results in the problems sketched in the introduction.

7.1.3 Correction

The correction is simple in this case. We must ensure that the irreflexivity property is valid for `red` and `green` as well. Since we used pattern-matching in the implementation, we only have to add the patterns for the cases `red < red` and

```
final proof state is:
<
[0]
targets: <>
premises: <§0 [(red < red)], §1 [(red < green)],
§2 [(red < blue)], §3 (green < red) === false,
§4 (green < green) === IF false THEN false ELSE true FI,
§5 (green < blue) === IF false THEN false ELSE true FI,
§6 (blue < red) === false, §7 (blue < green) === false,
§8 (blue < blue) === false, §9 [(red < red)]>,
[1]
targets: <>
premises: <§0 [(red < red)], §1 [(red < green)],
§2 [(red < blue)], §3 (green < red) === false,
§4 (green < green) === IF false THEN false ELSE true FI,
§5 (green < blue) === IF false THEN false ELSE true FI,
§6 (blue < red) === false, §7 (blue < green) === false,
§8 (blue < blue) === false, §9 [(green < green)]>>
end of final proof state
```

Figure 7.2: The Final Proof State of the Irreflexivity Proof

green < green. The best-fit pattern-matching of OPAL will create the correct implementation.

Program 7.6 The Ordering on Colours - Corrected

```
IMPLEMENTATION ColourOrd

  DEF red < red     == false
  DEF red < x       == true
  DEF green < red   == false
  DEF green < green == false
  DEF green < x     == true
  DEF blue < x      == false
```

7.2 The Deque Example

Next, we study the failed implementation of the deque data type. Deques are algebraically equivalent to sequences, but their run-time behaviour is different. Deques allow (amortized) $O(1)$ access to the last element as well as to the first element.

The example in Program 7.7 has been augmented with respect to Program 1.4 in the introduction. The `exist?` function receives two arguments, a predicate and a deque, and returns true, iff there is at least one element in the deque that fulfils the predicate. The `find?` function is an extension of `exist?`: if there are elements that fulfil the predicate, one of them is selected. The `in` function used in the specifications of `exist?` and `find?` is omitted for lack of space.

The implementation defines deques as an abstraction of a pair of sequences (Line 4). The CONG keyword in Line 5 is explained immediately in Section 7.2.1. Lines 6-11 define the operations induced by the free type declaration in terms of the implementation. The implementations of `exist?` and `find?` (Lines 16-21) make use of the corresponding functions on sequences. Of course, we could have used the simple definition DEF `exist?(P, d) == exist?(P, asSeq(d))`, but the implementation given here is more efficient, because we spare one call to the concatenation function and one to the `revert` function.

7.2.1 Data-Type Implementation

The data-type implementation of deques uses multiple representations. To make this known to the compiler the declarations in Line 5 and the function definition of `asSeq` (Lines 14-15) are necessary.

Line 5 shows an extension to regular OPAL. The declaration CONG `deq: asSeq` has the semantics that two deques are considered to be observationally equal, if both are mapped to identical values by `asSeq`, i. e. `x` $\cong_{deq}$ `y` **iff** `asSeq(x)` $\equiv$ `asSeq(y)`. (This means that the equivalence predicate demanded in Section 3.3.6 is ALL `x y. asSeq(x)` $\equiv$ `asSeq(y).`) This declaration entails proof obligations for every function from the interface that contains the type `deq` in its functionality. Those functions must not distinguish between different representations of observationally equal implementations.

We are interested in the congruence properties of `exist?` and `find?`, shown in Figure 7.3.

`Cong[exist?]`
ALL `a b b'.(asSeq(b)` $\equiv$ `asSeq(b'))` $\Longrightarrow$ `(exist?(a, b)` $\equiv$ `exist?(a, b'))`
`Cong[find?]`
ALL `a b b'.(asSeq(b)` $\equiv$ `asSeq(b'))` $\Longrightarrow$ `(find?(a, b)` $\equiv$ `find?(a, b'))`

Figure 7.3: Congruence Properties

A data-type implementation may also make use of junk elements, i. e. elements that do not correspond to any of the elements of the interface. We will deal with

Program 7.7 Deques

```
SIGNATURE Deq[α]                                                    1
  IMPORT Option[α] ONLY option avail? nil? cont                     2

  SORT α                                                            3
  TYPE deq == ◊                                                     4
              ::(ft: α, rt: deq)                                    5

  FUN exist?: (α → bool) × deq → bool                               6
      SPC  exist?(P,d) == r                                         7
      PRE  true                                                     8
      POST r ⇔ EX x. P(x) AND x in d                                9

  FUN find?: (α → bool) × deq → option[α]                           10
      SPC  find?(P,d) == r                                          11
      PRE  true                                                     12
      POST (avail?(r) ⟹ cont(r) in d AND P(cont(r))) AND            13
           (nil?(r) ⟹ ALL x. x in d ⟹ NOT P(x))                    14
  ...                                                               15
```

```
IMPLEMENTATION Deq[α]                                               1
  IMPORT Seq[α] ONLY seq ◊ :: ft rt ◊? ::? ⊣⊢                       2
                     revert exist? find? last                       3

  DATA deq == abs(left: seq[α], right: seq[α])                      4
  CONG deq: asSeq                                                   5

  DEF ◊ == abs(◊, ◊)                                                6
  DEF x::(abs(l,r)) == abs(x::l,r)                                  7
  DEF ft(abs(x::l,r)) == x                                          8
  DEF ft(abs(◊,r))    == last(r)                                    9
  DEF rt(abs(x::l,r)) == abs(l,r)                                   10
  DEF rt(abs(◊,r))    == abs(rt(revert(r)), ◊)                      11

  DEF ◊?(abs(l,r)) == (l◊?)  and  (r◊?)                             12
  DEF ::?(abs(l,r)) == (l ::?)  or  (r ::?)                         13

  FUN asSeq: deq → seq[α]                                           14
  DEF asSeq(d) == left(d) ⊣⊢ revert(right(d))                       15

  DEF exist?(P,d) ==                                                16
      IF exist?(P, left(d)) THEN true                               17
                            ELSE exist?(P, right(d)) FI             18

  DEF find?(P,d) ==                                                 19
      IF avail?(find?(P, left(d))) THEN find?(P, left(d))           20
                                   ELSE find?(P, right(d)) ⁉ FI     21
```

junk in Section 7.3; the implementation of deques does not make use of junk elements.

7.2.2 Proof Obligations

The proof obligations for the implementation of deques (Program 7.7) are shown in Figure 7.4. The proof obligations fall into three groups:

- the lifted formulas from the interface
- the closedness properties
- the congruence properties

```
Proof obligations:
{Lift[Dfd[exist?]],Lift[Dfd[find?]],Lift[Spc[exist?]],
 Lift[Spc[find?]],Lift[Freetype[deq]],
 Closed[::],Closed[rt], Closed[<>],
 Cong[::],Cong[::?],Cong[rt],Cong[ft],Cong[<>?],
 Cong[exist?],Cong[find?]}
```

Figure 7.4: The Proof Obligations for the Implementation of Deques

In this example, most proof obligations can be justified. This includes the specifications of `exist?` and `find?`. So we leave out the justifications of these proof obligations and concentrate on the justifications for the congruence properties `Cong[exist?]` and `Cong[find?]`. `Cong[exist?]` is valid, but the justification of this proof obligation helps in understanding the mistake in the definition of `find?`.

7.2.3 Justification

We plan to perform the justification by a formal proof. We start with the simpler task of justifying `Cong[exist?]`. Program 7.8 shows the proof declaration and the beginning of the proof tactic. The proof declaration uses the definitional equations, several definedness properties and some properties of the `exist?` function on sequences. The properties are shown in Program 7.9.

The concatenation properties allow to add or remove parts of the sequence. The two variants of LAW `exist_conc2` are useful in different parts of the proof. Actually, they are not equivalent: `exist_conc2a` follows from `exist_conc2b`. Finally, LAW `exist_revert` states that the result of applying `exist?` is independent of a previous `revert`.

Program 7.8 The Justification of cong[exist?]

```
IMPORT SeqLaws[α] COMPLETELY

PROOF cong_exist : Def[asSeq] Def[exist?] dfd_exist dfd_revert
                   SDfd[left] SDfd[right] exist_conc1
                   exist_conc2a exist_conc2b exist_revert ⟹
                   Cong[exist?]
JUSTF cong_exist == FORMALPROOF (rewrite_r_deep; rewrite_r_deep; ...
```

Program 7.9 Some Properties of exist? on Sequences

```
LAW exist_conc1  == ALL s1 s2 P. exist?(P, s1) ⟹ exist?(P, s1 ⊣⊢ s2)
LAW exist_conc2a == ALL s1 s2 P. exist?(P, s1) ≡ false AND
                                 exist?(P, s1 ⊣⊢ s2) ⟹ exist?(P, s2)
LAW exist_conc2b == ALL s1 s2 P. exist?(P, s1) ≡ false ⟹
                                 exist?(P, s1 ⊣⊢ s2) ≡ exist?(P, s2)
LAW exist_revert == ALL s P. exist?(P, s) ≡ exist?(P, revert(s))
```

The implementation of find? is similar to the implementation of exist?, and therefore we could expect that the justification of Cong[find?] might be easily derived from the justification of Cong[exist?]. In collecting the premises for a proof declaration cong_find, we detect that this is not the case. The concatenation properties are valid for find? as well (with some small changes, because exist? returns a Boolean value, and find? returns an option), but the result of find? for a reversed sequence is in general different from the result of find? for the original sequence. So there is no law find_revert that corresponds to the formula exist_revert.

7.2.4 Correction

Without actually starting the proof, merely by inspecting the proof for a similar function, we found a mistake in the implementation. The correction is simple: If we cannot use the identity find?(P, s) ≡ find?(P, revert(s)) during the proof, we have to revert the sequence in the implementation. Program 7.10 shows the correct implementation of the find? function. Note that this implementation is still more efficient than DEF find?(P, d) == find?(P, asSeq(d)), because asSeq contains an additional call to the concatenation function.

Program 7.10 A Correct Implementation of find?

```
DEF find?(P, d) ==
    IF avail?(find?(P, left(d))) THEN find?(P, left(d))
                                 ELSE find?(P, revert(right(d))) FI
```

7.3 The Colour Data Type

We continue the series of examples with the implementation of a data type `colour`. This example is admittedly artificial. The purpose of this example is twofold: On the one hand, the example serves to illustrate the implementation of a data type with junk elements. On the other hand, this example illustrates that it is not always enough or possible to prove the inclusion `Incl[f]` of a formula `f`. Instead, it is necessary to prove `Lift[f]` (see Sections 3.3.5 and 5.4).

Program 7.11 shows the interface unit `Colour.sign` and the implementation unit `Colour.impl`, which contain a data-type declaration and the declaration of a function `inv` together with some properties. The `inv` function is specified by three properties: It is its own inverse (LAW `inv_inv`), it has a fixpoint (LAW `inv_fix`), and the only fixpoint is `green` (LAW `inv_fix_green`)[1].

The `colour` data type is implemented by triples of real numbers. So we must implement the constructors and discriminators from the interface in terms of the implementation. This is done in Lines 4-6 and Lines 7-9 respectively. Line 10 contains the definition of the `inv` function.

Program 7.11 The Colour Data Type

```
SIGNATURE Colour                                                        1

  TYPE colour == red green blue                                         2

  FUN inv: colour → colour                                              3
  LAW inv_inv       == ALL x. inv(inv(x)) ≡ x                           4
  LAW inv_fix       == EX x. inv(x) ≡ x                                 5
  LAW inv_fix_green == ALL x. inv(x) ≡ x ⟹ x ≡ green                    6
IMPLEMENTATION Colour                                                   1
  IMPORT Real ONLY real 0 1 − =                                         2

  DATA colour == rgb(r: real, g: real, b: real)                         3

  DEF red   == rgb(1,0,0)                                               4
  DEF green == rgb(0,1,0)                                               5
  DEF blue  == rgb(0,0,1)                                               6

  DEF red?(rgb)   == r(rgb) = 1 and g(rgb) = 0 and b(rgb) = 0           7
  DEF green?(rgb) == r(rgb) = 0 and g(rgb) = 1 and b(rgb) = 0           8
  DEF blue?(rgb)  == r(rgb) = 0 and g(rgb) = 0 and b(rgb) = 1           9

  DEF inv(rgb(r,g,b)) == rgb(1 − r, g, 1 − b) ∎                         10
```

[1] These properties uniquely determine the `inv` function.

7.3.1 Proof Obligations

The proof obligations (as shown by the command `jcheck-info env`) are these:

```
Proof obligations:
Lift[LAW inv_inv],Lift[LAW inv_fix],Lift[LAW inv_fix_green],
Lift[Freetype[colour]],Closed[blue],Closed[green],Closed[red],
Closed[inv]
```

Figure 7.5: The Proof Obligations of Program 7.11

The proof obligations fall into two parts. The closedness properties ensure that no junk is visible and the lifted formulas ensure that the implementation meets the requirements from the interface. We explain these obligations in the following two sections.

7.3.1.1 Representation of Colour

The implementation of a data type may contain "junk", i. e. values that are not accessible with functions from the interface, and it may use multiple representations for the implementation of values. We have already seen the use of multiple representations in the previous example.

The representation of the `colour` data type uses junk elements. Since `colour` is a free type, the compiler is able to derive a predicate that describes the visible (non-junk) part of the `colour` data type. The set data type requires an explicit definition of the visible part, because `set` is not a free type. See Line 4 in the implementation of `set` in Program 7.1.

```
Colour.impl>jcheck-info sorts
colour: visible(x) <=> [red?(x)] OR [green?(x)] OR [blue?(x)],
no multiple representations
```

Figure 7.6: The Visibility Predicate for `colour`

The decision to implement the data type differently from the interface entails additional proof obligations (see Section 3.3.6). The user must prove that the functions in the interface are closed on the visible values. The closedness property for the `inv` function is shown in Figure 7.7.

```
Colour.impl>jcheck-info formula Closed[inv]
ALL a:colour. ([red?(a)] OR [green?(a)] OR [blue?(a)]) ==>
          ([red?(inv(a))] OR [green?(inv(a))] OR [blue?(inv(a))])
```

Figure 7.7: The Closedness Property for `inv`

7.3.1.2 Included vs. Lifted Formulas

The second purpose of this example is to illustrate the relationship between included and lifted formulas. The peculiar laws have been chosen for this purpose. In Figure 7.8 the inclusion and the lifting of the formulas from the interface are shown; `Lift[Freetype[colour]]` has been omitted, because its representation is too long. The underlined parts of the lifted formulas are those that restrict the domain of the bound variable to the visible values.

LAW inv_inv	
Incl	ALL x. inv(inv(x)) ≡ x
Lift	ALL x. ([red?(x)] OR [green?(x)] OR [blue?(x)]) ⟹ (inv(inv(x)) ≡ x)
LAW inv_fix	
Incl	EX x. inv(x) ≡ x
Lift	EX x. ([red?(x)] OR [green?(x)] OR [blue?(x)]) AND (inv(x) ≡ x)
LAW inv_fix_green	
Incl	ALL x. (inv(x) ≡ x) ⟹ (x ≡ green)
Lift	ALL x. ([red?(x)] OR [green?(x)] OR [blue?(x)]) ⟹ ((inv(x) ≡ x) ⟹ (x ≡ green))

Figure 7.8: Inclusion vs. Lifting

The relationship between included and lifted laws is different for the three formulas.

`inv_inv` This is the "standard" case. The implication `Incl[inv_inv]` ⟹ `Lift[inv_inv]` is valid[2] and the compiler detects this and adds an appropriate proof declaration. The user has two options: it is possible to justify either `Incl[inv_inv]`, which has the advantage of being syntactically equal to the law in the interface, or `Lift[inv_inv]`, which might be easier to justify (because the domain is restricted) but has a more complicated structure.

`inv_fix` The situation is different for the second law. The law `Incl[inv_fix]` is valid, because `inv` has fixpoints: all members of the set $\{(0.5, x, 0.5) | x \in R\}$ are fixpoints of `inv`. The proof obligation `Lift[inv_fix]` states explicitly that only fixpoints that are visible are to be considered. This proof obligation cannot be fulfilled. This property shows that we cannot always add a proof declaration `Incl[f]` ⟹ `Lift[f]` for an arbitrary formula `f`.

[2]The implication is always valid, if the only quantifier is a universal quantifier and no negation is present.

inv_fix_green Finally, LAW **inv_fix_green** states that if there is a fixpoint, then this fixpoint is equal to **green**. In this case, the restriction to the visible values is crucial. The formula **Incl[inv_fix_green]** is not valid in the implementation, but the lifted version **Lift[inv_fix_green]** is valid.

7.3.2 Justification Attempt

A first attempt to justify that the implementation fulfils the requirements listed in the interface unit is shown in Program 7.12. The formal proof for **inv_fix_green** requires some auxiliary lemmas and is omitted in this presentation. The introduction of LAW **inv_fix_i** eases the formal proof of LAW **inv_fix**.

Program 7.12 The Justification Attempt for inv-Related Proof Obligations

```
PROOF inv_closed: ⟹ Closed[inv]
JUSTF inv_closed == FORMALTEST (inv, allGuards, %(red, green, blue))

PROOF inv_inv: ⟹ Incl[inv_inv]
JUSTF inv_inv == FORMALTEST (inv, allGuards, %(red, green, blue))

LAW inv_fix_i == inv(green) ≡ green
PROOF inv_fix_i: Def[inv] Def[green] dfd_0_real dfd_1_real ⟹ inv_fix_i
JUSTF inv_fix_i == FORMALPROOF (rewriter30)

PROOF inv_fix: inv_fix_i Lift[Discr[green, green?]] ⟹ Lift[inv_fix]
JUSTF inv_fix == FORMALPROOF (ex_r; tApartS)

PROOF inv_fix_green: redX greenX blueX no_fix ⟹ Lift[inv_fix_green]
JUSTF inv_fix_green == FORMALPROOF ...
```

The error messages shown by the compiler point directly to the problematic issues: The first error message indicates that the justification for **inv_closed** failed for input data **green**. The result is "some" unprintable value. The second error message shows an unresolved subgoal of the formal proof of **inv_fix_i**, which could be simplified further, but one can already recognize that the proof will not succeed.

```
ERROR [Colour.impl at 81.7-81.16]: test failed for input
                                   data no. 1 <some>
ERROR [Colour.impl at 88.7-88.15]: unfinished proof
  unresolved subgoals are:
  <rgb((1 - 0),1,(1 - 0)) === rgb(0,1,0)>
```

Figure 7.9: The Error Messages of the Failed Justification Attempt

Both errors have the same reason: the `inv` function returns a wrong colour. Neither holds the closedness property, nor is `green` a fixpoint of `inv`:
`inv(green)` $\rightsquigarrow$ `inv(rgb(0, 1, 0))` $\rightsquigarrow$ `rgb(1 − 0, 1, 1 − 0)` $\rightsquigarrow$ `rgb(1, 1, 1)` $\rightsquigarrow$?

This error can lead to cryptic behaviour. The function `print` in Figure 7.13 converts a value of type `colour` to a readable representation.

Program 7.13 Printing Colours

```
FUN print : colour → denotation

DEF print(red)   == "red"
DEF print(green) == "green"
DEF print(blue)  == "blue"
```

Evaluating the expression `print(inv(green))` will result in a run-time error message that complains about a "missing else in print". The OPAL compiler uses a defensive translation scheme that inserts an additional else branch that generates the error message. If the compiler is instructed to optimize aggressively, no else branch is added, and the result of `print(inv(green))` will be an arbitrary value, which makes search for the error even more difficult.

7.3.3 Correction

The correction is not as obvious as it was in the previous examples. Still, it is not too difficult to come up with a correct implementation. Instead of inverting the red and blue components separately, we swap the red and blue components (see Program 7.14). After this change, all of the above verification tasks go through.

Program 7.14 The Colour Implementation - Corrected

```
IMPLEMENTATION Colour
  IMPORT Real ONLY real 0 1 − =

  DATA colour == rgb(r : real, g : real, b : real)

  DEF red   == rgb(1, 0, 0)
  DEF green == rgb(0, 1, 0)
  DEF blue  == rgb(0, 0, 1)

  DEF red?(rgb)   == r(rgb) = 1 and g(rgb) = 0 and b(rgb) = 0
  DEF green?(rgb) == r(rgb) = 0 and g(rgb) = 1 and b(rgb) = 0
  DEF blue?(rgb)  == r(rgb) = 0 and g(rgb) = 0 and b(rgb) = 1

  DEF inv(rgb(r, g, b)) == rgb(b, g, r)
```

7.4 Other Languages

To round off our presentation of examples, we conclude with two sketches in languages other than OPAL/J. Both sketches deal with the problem of defining an ordering relation on the `colour` data type. The examples are not fully equivalent to Program 7.3 because we try to stay close to the spirit of the respective language.

The first of these sketches is written in HASKELL [PH99]. HASKELL is a general purpose, purely functional programming language like OPAL and ML, but with some differences like lazy evaluation and type classes.

The second example is written in JAVA. With JAVA we present how a "literate justification" might look in a non-functional programming language.

7.4.1 Haskell: Type Classes with Specifications

In HASKELL type classes are used to introduce overloaded functions in a restricted, structured way [WB89]. A type-class declaration introduces the names of the overloaded functions that must be supported by every type that is an instance of this type class. Types can be declared to be instances of type classes. An instance declaration includes definitions of the overloaded functions declared in the type class.

Program 7.15 shows the type classes `Eq` for types that have an equality function, and `Ord` for types that have a total order[3]. Both type classes provide default definitions for some functions. For example, type class `Eq` provides a default definition for the `/=` (unequal) function. In the definition of `Ord` the functions `<=`, `<`, `>=` and `>` are defined in terms of the `compare` function. For brevity we omit the default definition of `compare` in terms of `==` and `<=` and the default definition of `max` and `min`. The user must define either the `compare` function or the equal and less-than function in order to make sensible use of the class `Ord`.

In principle there is no semantic meaning associated with type classes and the overloaded functions. Of course we do associate certain algebraic properties with a function named "<" and therefore it makes sense to add specifications to the source code to make these hidden assumptions visible. Program 7.16 shows the `Ord` type class with an additional specification of the `compare` function. The laws are introduced by their name, followed by a formula.

The implementation of the `Colour` data type in HASKELL is shown in Program 7.17. The first line declares the data type and the constructor functions, followed by the definition of the function `cmp` and two instance declarations. The

[3]Note that HASKELL uses == and :: where OPAL uses = and : and vice versa.

Program 7.15 HASKELL: The Eq and Ord Type Classes

```
class Eq a where
      (==), (/=)  :: a -> a -> bool
      x /= y      =  not (x == y)

class (Eq a) ==> Ord a where
       compare                  :: a -> a -> Ordering
       (<), (<=), (>=), (>)     :: a -> a -> Bool
       max, min                 :: a -> a -> a
 ...
   x <= y              = compare x y /= GT
   x <  y              = compare x y == LT
   x >= y              = compare x y /= LT
   x >  y              = compare x y == GT
...
```

Program 7.16 HASKELL: The Ord Type Class with Specification

```
class (Eq a) ==> Ord a where
       compare                  :: a -> a -> Ordering
       (<), (<=), (>=), (>)     :: a -> a -> Bool
       max, min                 :: a -> a -> a

   compare x y
           | x == y    = EQ
           | x <= y    = LT
           | otherwise = GT

   dfd        :: all x y -> dfd compare x y
   reflexive  :: all x y -> compare x y === EQ <=> x === y
   symmetric  :: all x y -> compare x y === LT <=> compare y x === GT
   transitive :: all x y z -> compare x y === LT &&
                              compare y z === LT ==> compare x z === LT
```

function cmp need not be declared separately, its type is inferred from the definitions.

Program 7.17 HASKELL: The Colour Data Type with Justifications

```
data Colour = Red | Green | Blue

cmp Red Red = EQ;   cmp Green Red = GT;     cmp Blue Red = GT;
cmp Red Green = LT; cmp Green Green = EQ;   cmp Blue Green = GT;
cmp Red Blue = LT;  cmp Green Blue = GT; ⁉  cmp Blue Blue = EQ;

instance Eq  Colour where x == y = cmp x y == EQ
instance Ord Colour where compare = cmp

DfdColour   :: Def cmp ==> dfd
DfdColour   = FormalProof {mInduct; haskellR; tApartS}
ReflColour  :: Def cmp ==> reflexive
ReflColour  = FormalProof {mInduct; haskellR; tApartS}
SymColour   :: Def cmp ==> symmetric
SymColour   = FormalProof {mInduct; haskellR; tApartS}
TransColour :: Def cmp ==> transitive
TransColour = FormalProof {mInduct; haskellR; tApartS}
```

The instance declaration entails proof obligations for the laws of the specification. The lower half of Program 7.17 shows the additional proof declarations and justifications. HASKELL does not require to declare functions, but we have to add proof declarations explicitly, because the unit correctness check depends on them. `Def cmp` refers to the definition of the `cmp` function. The syntax of the justifications assumes that `FormalProof` introduces a sequence of monadic commands like the HASKELL keyword `do`.

The definition of `cmp` violates the symmetricity law, because `cmp Green Blue` and `cmp Blue Green` both evaluate to `GT`.

7.4.2 Java: Interfaces with Specifications

For JAVA we want to show how the ordering of the colour data type could be checked. We have already given the definition of the `Comparable` interface in Section 2.1.3, where the method `compareTo` is specified informally. Program 7.18 shows how the interface could be specified formally. We assume that the identifiers declared within the laws are universally quantified (and avoid answering the question how existential or nested quantifiers should be written down explicitly).

Program 7.18 JAVA: The `Comparable` Interface – with Specification

```
public interface Comparable {
  public int compareTo(Object o)
  /** Compares this object with the specified object for order. Returns
  a negative integer, zero, or a positive integer as this object is less than,
  equal to, or greater than the specified object.

  It is strongly recommended, but not strictly required that
  (x.compareTo(y)==0) == (x.equals(y)). Generally speaking,
  any class that implements the Comparable interface and violates
  this condition should clearly indicate this fact. The recommended
  language is "Note: this class has a natural ordering that is
  inconsistent with equals."
  */
    public law symmetric(Object x, Object y) {
      sgn(x.compareTo(y)) == -sgn(y.compareTo(x))
    }
    public law transitive(Object x, Object y, Object z) {
      (x.compareTo(y) > 0 && y.compareTo(z) > 0)
          implies x.compareTo(z) > 0
    }
    public law equivalent(Object x, Object y, Object z) {
      x.compareTo(y) == 0 implies
          (sgn(x.compareTo(z)) == sgn(y.compareTo(z)))
    }
}
```

The colour data type is defined as a JAVA object in Program 7.19. JAVA does not support enumeration types very well, therefore we defined the comparison method in terms of the internal representation.

We tried to make the justification syntax fit into JAVA's syntactic conventions. The justifications start with the "type" of the justification, followed by the keyword `justf`. This is followed by the target law, in turn followed by the premises (the definition of the method `compareTo` and laws on the `double` type). Finally the specific information for the justification is given. In our example, we use always the same tactic: we start with an induction over the objects accessible to the public, apply some JAVA specific rules, and use simple logic to resolve the proof.

The proofs of the transitivity law and the equivalence succeed, but the proof of the symmetricity law fails. The result will be similar to the result in Section 7.1.2, but this time the formula `sgn(red.compareTo(green)) ==`

Program 7.19 JAVA: The Colour Data Type

```
class Colour implements Comparable {
  private double red;
  private double green;
  private double blue;
  private Colour(double red, double green, double blue) {
          this.red = red; this.green = green; this.blue = blue;}
  public static Colour Red   = new Colour(1, 0, 0);
  public static Colour Green = new Colour(0, 1, 0);
  public static Colour Blue  = new Colour(0, 0, 1);

  public int compareTo(Colour c) {
    return (int) ((this.red + this.green + 2 + this.blue * 4) -
                  (c.red + c.green * 2 + c.blue * 4));

  formal proof justf symmetric(compareTo, doubleLaws)
        { mInduct; javaR; tApartS; }
  formal proof justf transitive(compareTo, doubleLaws)
        { mInduct; javaR; tApartS; }
  formal proof justf equivalent(compareTo, doubleLaws)
        { mInduct; javaR; tApartS; }
  }
}
```

`-sgn(green.compareTo(red))` will indicate the error committed in the implementation.

Chapter 8

Security and Correctness Checks

The introduction of an interpreter as a tool in the compilation process causes a security risk. The interpreter is used for the implementation of external tools and for the execution of tested functions. The implementation of external tools with the help of an interpreter poses only a low risk. Of course the implementation of these tools might contain mistakes, but in principle these tools can be trusted as much as the compiler itself can be trusted.

But the execution of functions always involves a security risk, and even more so, if the function was implemented by an unknown or untrusted programmer. So it might happen that the execution of the tested function with some of the test-data values has a malicious effect on the host system, such as deleting files or compromising the user's security.

So far we have ignored this issue, because we have concentrated on *pure* functional languages, i. e. languages without side effects. If the compiler works properly, the execution of pure functions cannot have malicious effects. But if our approach is transferred to a language with side effects, we must expect that programs expose bad behaviour.

Options to ensure that execution of untrusted software poses no risk fall into two categories: We may either prove before execution that the untrusted code is secure, or we can monitor the program while it is executed in order to prevent it from doing real damage.

We present the sandboxing technique, the JAVA byte-code verifier and the Proof-Carrying Code approach, and then discuss how we can lower the security risks in the OPAL/J compiler.

8.1 Sandboxing

The technique of monitoring the execution of the program is also known as "sandboxing". The sandbox metaphor describes the restricted access of the program to the outer world. Any access to the area outside the sandbox (e. g. to the host's file system) is considered a harmful operation and inhibited.

Sandboxing is most easily implemented by interpreting the untrusted code. In this case the interpreter is extended to not fulfil requests for dangerous operations by untrusted sources. Another possibility is to insert run-time checks into native code. While this has to be done at compile time, modern operating systems offer a possibility to add checks dynamically by intercepting calls to functions from dynamic libraries [BG99].

Sandboxing has become known as the technique that ensures security of JAVA code, but has been employed before in multitasking environments. Processes running in parallel should not interfere, but buggy programs might access or change memory that belongs to other processes. The operating system must protect user and – even more important – system processes from other, possibly hostile, programs. The UNIX operating system allows to limit the memory size, the CPU time of a process or the maximum size of a file that a process may create.

The advantage of the sandboxing technique is its feasibility. The disadvantages are the overhead involved, because every instruction must be checked before it is actually executed, and its coarse security model that excludes many applications.

8.2 The Java Byte-Code Verifier

The JAVA byte-code verifier is called whenever an untrusted applet is loaded. The implementation as described in [LY97] splits the verification into four passes, which are performed at different stages during loading, linking and execution of an applet. Since it may not be assumed that the class file was generated by a (friendly) JAVA compiler, the verifier must repeat some of the work that another compiler already did.

Pass 1 roughly corresponds to a syntax check. When a class file is loaded, it is checked for its basic format: correct magic number, proper length of attributes, and so on. After this pass, the file is known to adhere to the class file structure.

Pass 2 is executed, when the class file is linked. During Pass 2, checks are made that can be done without looking at the code array: the class structure must be well-built and the constant pool is checked for consistency.

Pass 3 finally looks at the code. First the code is checked for consistency: branches are only allowed within the same method, branches may not target the middle of an instruction, and other conditions. Then, the typing of operands, method invocations and field accesses are checked, as well as proper access to local variables and the operand stack, which must not over- or underflow.

The final Pass 4 is done when a type, method or field is used for the first time. Only then the access permission and the existence of the method or the field in the given class is checked.

The notion of "verification" seems exaggerated, since there are no formally defined properties that are checked[1]. The only property that is actually checked is the definedness of the (operational) semantics of the object code. From the algebraic point of view, this is not an exciting property. For systems security this is of course great progress. Many security holes exploit undefinedness properties, e. g. accessing memory that does not belong to the own process, or deliberately overflowing the stack.

The byte-code verifier checks "low-level" security properties, properties that can be expressed on the object level. Security properties that deal with access to certain files or services are not checked by the verifier, but are dealt with by the sandboxing technique of the JAVA byte-code interpreter.

8.3 Proof-Carrying Code

Proof-Carrying Code [Nec97] is a recently developed mechanism by which a host can determine whether code is safe to execute. The typical scenario consists of the host, which will only execute code that adheres to the host's safety policy, and of the programmer, who writes a program that is to be executed by the host. Normally, it is the host, who must check that execution of the program is secure. But without knowledge of the source code, it is very difficult to perform a proof of security.

In the Proof-Carrying Code approach, it is the programmer, who generates a "safety proof" [sic!] in addition to the object code, possibly with the help of the compiler. This proof is bundled with the object code and sent to the code consumer. This corresponds to a certification, but certification only establishes the authorship and relies on personal authority for the proof. In contrast, a safety proof has nothing to do with authorship, but provides a greater degree of correctness.

[1] Once in [LY97] a "simple theorem-prover" is mentioned, but its activities during the verification process are not further detailed.

The code consumer can independently verify that the code obeys the necessary safety properties. This verification is cheap because the proof has already been produced. The verification process consists of mere proof validation: it checks first, whether the proof is indeed a proof for the safety properties and second, whether the proof matches the object code. The verification can also be trusted, because the host need only execute a simple verifier of his own choice.

Some technical points have to be considered, e. g. the representation of proofs [NL98b]. The approach has been used in several case studies, one of them being the extension of ML by code written in C that does obey the ML type constraints [Nec97].

8.4 Comparison of Byte-Code Verification and Proof-Carrying Code

Both approaches, JAVA byte-code verification and Proof-Carrying Code, perform their respective verifications on the object code; the source code is not available. The checks performed necessitate the addition of information to the object code about types, or objects, or loop invariants. This is a lot of valuable information about the source code – some JAVA byte-code disassemblers (MOCHA) do a frighteningly good job in reconstructing source code from byte-code. It seems that verification even of restricted areas needs most information of the source code[2].

The approach used by the JAVA byte-code verifier has the disadvantage that the whole work must be performed by the host. The code producer is not required in any way to help the verifier. Consequently, the information gathered by the verifier is not as specific as in the case of Proof-Carrying Code. The JVM may assume that the code does not compromise the security of the operating system. High-level security is not handled by the byte-code verifier (but dealt with by the sandbox approach).

The combination of a byte-code verifier and a sandbox interpreter is not enough to guarantee secure execution. The code in Program 8.1 (published in [Van96]) demonstrates how ignoring implicit assumptions about methods can be exploited for evil purposes. The `applet` class is defined with a (non-final) method `stop`, intended to clean up after an applet exits, for example to kill threads created by the applet. The hostile applet overrides the method `stop` with an empty body. The JVM executing this applet has no means to stop the denial-of-service attack started by this hostile applet.

[2] This development raises interesting questions concerning the protection of intellectual property.

Program 8.1 A Part of `Consume.java`, an Hostile Applet

```
/*  Create and start the offending thread in the standard way */

    public void start() {
        if (wasteResources == null) {
        wasteResources = new Thread(this);
        wasteResources.setPriority(Thread.MAX_PRIORITY);
        wasteResources.start();
        }
    }

/*  We won't stop anything */

    public void stop() {}
```

The Proof-Carrying Code approach offers a framework to deal with a denial-of-service attack. What is needed to defend against this kind of attack, is a formal specification of the effects of `stop` in the `Applet` class, and a proof that this specification is met by every overriding method.

Another advantage of the Proof-Carrying code approach is that the host has only the cheap duty to check whether the proof supplied with the object code is indeed a safety proof for the accompanying program. The proof is developed only once. The JVM on the other hand will perform the byte-code verification every time the program is executed.

The partition into a proof and the object code provides another level of security. It is not enough to change either the proof or the object code, both must be changed consistently.

8.5 Application to an Integrated Interpreter

We must be able to handle two kinds of security problems that may occur using the interpreter.

- A *denial-of-service* attack: The function executed might not terminate or use too much memory or both.
- Execution of a *malicious function*: The malicious function could delete files, compromise data bases, etc.

We can deal with a denial-of-service attack by restricting the (time and space) resources we give to the interpreter. But if we restrict the resources too much, we will not be able to execute tested functions or to execute proof programs that do not expose bad behaviour but simply require more time or more memory for

a successful execution. We have to find a balance such that (almost) all harmless executions can be carried out, but harmful executions are stopped before wasting too much resources.

Malicious functions can be executed in a sandbox. Since sandboxing lowers the performance, only user-supplied functions should be run within the sandbox, but not the external tools, such as the testing component or the theorem-prover.

The Proof-Carrying Code approach is not directly applicable, because in our approach, it is the source code that carries proofs. The general model, however, can be used in our framework as well. The customer provides a formal specification of the desired safety properties as a part of the interface specification. The programmer provides a formal proof that the safety properties hold. The customer can use the integrated compiler to check the safety properties.

If the source code should not be disclosed to the customer, we have to bring in a trusted certification agency. The programmer sends the source code with literate justification to the trusted certification agency. The certification agency checks the safety properties. If the check succeeds, the certification agency certifies the object code (e. g. by a digital signature) and returns this certified object code to the programmer, who may sell the certified code without revealing the source code.

If we follow this approach, the compiler would have to be extended to treat the safety properties differently from other properties. First, the only justification method that is acceptable for safety properties, is a formal proof. Second, the safety properties must be completely proven correct before any test is carried out.

Chapter 9

Related Work

9.1 Languages

There are lots of programming languages and specification languages, but only a few of them have actually been defined with the verification process in mind. This includes most specification languages that have a clear semantic definition of the specification constructs and also define correctness conditions. But this alone makes verification not necessarily easy, because the formulation of the correctness conditions is not tailored towards easy verification.

We introduce two languages that have been designed as verifiable extensions to existing programming languages.

9.1.1 Euclid

The language EUCLID [LHL+77] is included here for historical reasons. EUCLID was derived from PASCAL "to make it more suitable for verification [...]."[PHL+77] and it is this design objective that makes the language interesting for our purposes. The authors "expect many of these changes [from PASCAL to EUCLID] to improve the reliability of the programming process, firstly by enlarging the class of errors that can be detected by the compiler, and secondly by making explicit in the program text more of the information needed for understanding and maintenance." The transfer of the verification process from the programmer to the language and its compiler [PHL+77] is seen as a natural continuation of the development of compilers.

The language itself is as close to PASCAL as possible; it was a design decision to change PASCAL only where necessary, e. g. visibility of names, pointer handling. A major addition is the introduction of "modules", which serve in EUCLID to

introduce abstract data types. For verification purposes specifications and assertions can be inserted into a program. Boolean expressions are used as the basic assertion language, other elements of the assertion language are defined by the particular verifier. These extended assertions are inserted as comments and thus hidden from the compiler. Proof rules for EUCLID using a Hoare-style axiomatic method have been published in [LGH+78]. One example procedure from the paper contains pre- and postconditions as part of a revised syntax. The correctness of the following procedure is proven in an appendix of [LGH+78].

```
procedure p(var a: signedInt, b: signedInt)=pre true; post a ≤ 2 * b;
   begin var c: signedInt; c := 2 * b; if a > b then a := c end if end.
```

The language authors' confidence in formal verification earned them an attack in [MLP79] as "foremost verification adherents", especially their decision not to include exceptions in EUCLID – because "we expect all Euclid programs to be verified" and "runtime software errors should not occur in verified programs" (both [PHL+77]) – was target of polemic comments. The authors of [MLP79] see this as an example of the so-called "Titanic effect" (when failure does occur it is massive and uncontrolled): "Errors should not occur? Shades of the ship that shouldn't be sunk".

9.1.2 Extended ML

The work on EXTENDED ML ([KST94], for a gentle introduction see [KST97]) closely resembles the approach presented in this thesis. EXTENDED ML was specifically developed as a framework for the formal development of ML software systems. EXTENDED ML is therefore a wide-spectrum language suitable for expressing all stages in the development of a ML program from the first high-level specification to the executable program.

Just as EUCLID was designed as a minimal extension of PASCAL, EXTENDED ML was designed to be a minimal extension of ML. Syntactically, axioms are added to structures, e. g. to express partial orders (copied from [ST89]):

```
signature PO =
   sig
      type elem
      val le : elem * elem -> bool
      axiom le(x,x)
      axiom le(x, y) andalso le(y,x) => x=y
      axiom le(x, y) andalso le(y,z) => le(x,z)
   end
```

Axioms are expressions of type `bool`, the expression language is augmented with quantifiers and keywords to express properties like termination and whether an

evaluation raises an exception. Another extension is the introduction of placeholders that serve to omit concrete definitions for types and functions. These placeholders are replaced with concrete implementations during program development.

The semantics of EXTENDED ML consists of a static semantics for type-checking, a dynamic semantics for evaluation – as with ML – and a verification semantics for "verificating" [sic!] that the constraints imposed by the axioms are met. This semantics does not define the proof obligations induced by development steps.

The definition of an extension to ML for specification purposes turned out to be more difficult than anticipated [KS98]. From the very beginning most imperative features were excluded from EXTENDED ML (so it is not really an extension of ML, but of a sublanguage of ML), but the features that remain – e. g. exceptions – still pose problems in reasoning about ML. Other problems include the possible mixing of formulas and Boolean expressions and dealing with behavioural equivalence.

An intricate problem is caused by an interaction between the type inference algorithm of EXTENDED ML and the fact that the validity of formulas may depend on the type of quantification. Consider the following example:

```
type 'a dummy = bool
val b: 'a dummy = forall (x, y: 'a) => x == y
```

In this example, a type `'a dummy` (in OPAL, one would write "dummy[α]") is introduced that is isomorphic to the type `bool`. Then, a constant `b` of this type is defined; the value of this constant is determined by an axiom (remember that axioms are expressions of type `bool`). If the constant `b` is applied somewhere, its value depends on the type inferred for `'a`. The value of `b` might be `true` or `false`, and it is not always possible the reconstruct the exact type the type checker infers[1]. Fortunately, this situation is rarely encountered. EXTENDED ML handles this problem by regarding axioms that contain type-dependent expressions as not satisfied.

9.2 Verification Systems

Most proof systems are directed towards interactive proving of theorems written in a specification or formula language tailored towards their specific logic. There are some verification systems that allow the user to write specifications in a functional programming style and allow interactive reasoning about these functions.

[1] For details, see [KS98].

These systems support the development of proofs and allow the replay of previously conducted proofs. The specifications are not intended to be run, however, and the systems do in general not check verification conditions.

PVS and the KIV system have the most complete approach. They support a module system and manage proof obligations and proofs. ISABELLE/HOL and COQ have been written for different purposes, but can also be used to prove the correctness of programs. STEP is a tool for the computer-aided formal verification of reactive system, which is interesting for us, because it also supports different verification methods. A recently proposed development system for HASKELL cannot be classified due to lack of information.

9.2.1 PVS

PVS [COR+95] is a Lisp-based system for specification and verification of programs. PVS was not designed to prove programs correct, the purpose is rather to help "in the detection of errors as well as in the confirmation of 'correctness.'" To this end, PVS uses a rich type system, provides operators that are guaranteed to preserve consistency and finally contains an effective theorem-prover.

Type checker and proof checker support each other. The type checker generates type-checking conditions that stem from the use of predicate subtypes and dependent types that are then given to the proof checker. The (interactive) proof checker in turn uses the type checker to ensure the well-typedness of the user's inputs. The proof support has been designed in a way that the tedious work of proof is done by the system, while the system is still controlled by the user.

A small example specification taken from the tutorial:

```
sum: THEORY
 BEGIN
  n: VAR nat
  sum(n): RECURSIVE nat =
   (IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF)
   MEASURE (LAMBDA n:n)
  closed_form: THEOREM sum(n) = (n * (n + 1))/2
END sum
```

The measure function is used to generate a termination proof obligation. The system generates two proof obligations for this specification. The first is due to the fact that "-" is not a total function[2], the second expresses the termination property.

[2]Actually, PVS functions expresses partiality by dependent types. In principle, all PVS functions are total.

```
% Subtype TCC generated (line 7) for n - 1
  % unchecked
sum_TCC1: OBLIGATION (FORALL (n: nat): NOT n = 0 IMPLIES n -1 >= 0);

% Termination TCC generated (line 7) for sum
  % unchecked
sum_TCC2: OBLIGATION (FORALL n: nat): NOT n = 0 IMPLIES n - 1 < n);
```

9.2.2 KIV

The KIV (Karlsruhe Interactive Verifier) system [Rei95] is a tool designed for the formal development of correct software systems. The formal development starts with a formal specification written in a specification language that is similar to ASL[3]. The components of the specifications are refined into intermediate specifications or, finally, into program modules. The implementation is a collection of functional programs in a PASCAL-like notation. The composition operators are constrained in a way that ensures compositionality of correctness.

The proof obligations are expressed as sequents of Dynamic Logic (DL). DL extends ordinary predicate logic by formulas $\langle\pi\rangle\phi$ ("if π terminates, ϕ holds after execution of π") and $[\pi]\phi$ ("π terminates and ϕ holds after execution of π"). The following formula expresses the property that SUCC and PRED are inverse operations, if the condition *nlz* ("no leading zeroes") is valid for the input:

$$nlz(a) \vdash \langle \mathtt{SUCC}(a : b); \mathtt{PRED}(b : c)\rangle c = a$$

The main strength of the KIV system is the elaborated verification support. KIV pursues an evolutionary verification model that aids the development of a software module from the first erroneous version to a correct module. Changes made to either specifications or programs are analyzed for the effects to previous verification attempts. The failed proofs are reused in the construction of the new proof in the next development step. If this is not possible, the proof has to be completed interactively.

The KIV system has been used in several case studies and projects. A list of some recent applications is contained in [RSS97].

[3] ASL is a kernel specification language. See [Wir90, Section 9.3] for a short description and further references.

9.2.3 Isabelle/HOL

ISABELLE is a ML-based meta-tool for the development of provers that allows to implement one's own syntax for the object language and to define the rule system of the logic to be used in the proofs. Isabelle has been widely used to provide a proof environment for particular object logics. We have already discussed in Section 6.4.4.2 the role of ISABELLE as a reference implementation for the theorem-prover component of OPAL/J. The ISABELLE system has also been used for test-case generation in the ESPRESS project [HNS97].

In this section, we do not want to discuss the generic theorem-prover ISABELLE, we rather want to introduce ISABELLE/HOL [Nip98], which is one of several object-logics distributed with the ISABELLE environment (other logics are e. g. Zermelo-Fraenkel set theory or constructive type theory).

ISABELLE/HOL implements higher-order logic that can be used as an environment for specifying functional programs and verifying properties of these. The language of ISABELLE/HOL is similar to ML or HASKELL; it supports polymorphic types, data-type definitions and the usual constructions for expressions. As example, we present the definition of lists (see [Nip98]):

```
datatype 'a list = Nil                              ("[]")
                 | Cons 'a ('a list)                (infixr "#" 65)

consts app :: 'a list => 'a list => 'a list   (infixr "@" 65)

primrec
"[] @ ys       = ys"
"(x # xs) @ ys = x # (xs @ ys)"
```

A proof of associativity is short thanks to the sophisticated tactics supplied with the system. The user has to issue the following three commands:

```
Goal "(xs @ ys) @ zs = xs @ (ys @ zs)";
by(induct_tac "xs" 1);
by(Auto_tac);
```

It is possible to turn these commands into a single ML function that may be evaluated again, if the definition changes. There is no possibility, however, to automate this process. There are some syntactic (priority of `if-then-else`) and semantic properties (functions must be total) that make the unreflected use of ISABELLE/HOL for proving properties of programs problematic. Because ISABELLE/HOL can be changed and extended within the ISABELLE environment, it should be possible to derive an object-logic for a specific programming language on the basis of ISABELLE/HOL.

9.2.4 Coq

The Coq system [BBC+00a] is a proof assistant for the Calculus of Inductive Constructions. This proof assistant allows to extract programs from proofs via a strong extraction function, which forgets the non-computational parts of a proof. The result is a functional program in an ML dialect.

In [Par95a, Par95b] this approach is (almost) reversed to synthesize proofs from programs. The strong extraction function is non-reversible, but it is possible to define a weak extraction function that keeps enough information in the final program so that the reconstruction of the corresponding proof term is possible. The following example shows the result for a division algorithm that returns quotient and remainder:

```
let rec div a b =
match a with
   0  -> { (0 = b*0+0) ∧ (b > 0) }
         (0,0)
| n+1 -> { ∃ q,r(n = b*q+r) ∧ (b > r) → ∃ q,r(n+1 = b*q+r) ∧ (b > r) }
         let (q,r) = div n b in
            { (n+1 = b*q+r) ∧ (b > r) }
         if b<=(r+1)
         then { (n+1 = b*(q+1)+0) ∧ (b > 0) }
              (q+1,0)
         else { (n+1 = b*q+r+1) ∧ (b > r+1) }
              (q,r+1);;
```

The result resembles a program annotated with axioms in the Floyd-Hoare calculus, only this time a functional program is annotated. The similarity goes even further, e.g. there is a "weakest specification" that roughly corresponds to a weakest precondition. The approach has been implemented as a tactic in the Coq system.

9.2.5 STeP

The Stanford Temporal Prover, STeP [MBB+99], is a tool for the computer-aided formal verification of reactive systems, including real-time and hybrid systems, based on their temporal specification.

Figure 9.1 presents an outline of the STeP system. We recognize that the support of various verification systems plays a central role in the design of STeP: There are algorithmic, deductive-algorithmic and deductive methods available. According to the STeP tutorial [BBC+00b], "STeP is best viewed as providing a toolkit of verification methods [...]. A given system can be analyzed in a

number of ways. Depending on the system and property to be proved, different tools will be applicable or appropriate."

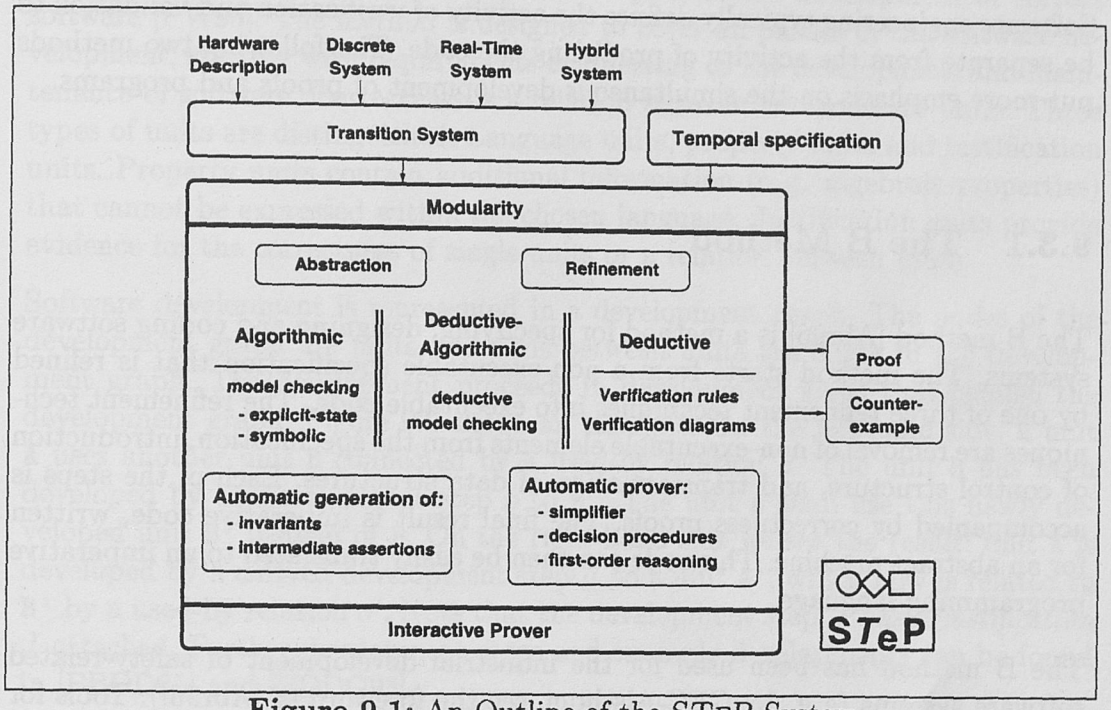

Figure 9.1: An Outline of the STeP System

Many of the components have been implemented as external components. This allows the use of different implementation languages for these components (C and ML) as well as the interaction of STeP with third-party theorem-provers.

9.2.6 Haskell

Recently, in [Jon00], a "new kind of program development environment" has been proposed that will "allow programmers to assert properties of program elements as part of their source code." The system is designed in a way that supports a "wide range of validation options", from automated testing to formal methods. A suite of "property management" tools will support the user in the management and development of validations.

The general objective of the system is quite similar to our approach. The description in [Jon00] is rather terse, a comparison to our approach is therefore difficult. The existence of separate property management tools suggests that the environment is formed by the aggregation of several tools, and not by integrating support for justifications into the compiler.

9.3 Software Engineering

Software engineering typically defines the activity of verification and validation to be separate from the activity of producing the code. The following two methods put more emphasis on the simultaneous development of proofs and programs.

9.3.1 The B Method

The B method [Abr96] is a method for specifying, designing and coding software systems. The method starts from a non-executable specification that is refined by one of three refinement techniques into executable code. The refinement techniques are removal of non-executable elements from the specification, introduction of control structure, and transformation of data structures. Each of the steps is accompanied by correctness proofs. The final result is imperative code, written for an abstract machine. This code can then be easily translated to an imperative programming language.

The B method has been used for the industrial development of safety-related software systems (e. g. by GEC-Alsthom, see the foreword of [Abr96]). Tools for software development with the B method are commercially available [B95]. The B-Book [Abr96] contains several examples, e. g. a steam-boiler control system; unfortunately the only properties used in these examples are typing invariants. The following piece of source code is part of the steam boiler example:

```
MACHINE
  Service_L_1
SEES
  Cycle_W_1, Cycle_S_1, Constants_SW_1
VISIBLE_VARIABLES
  llm, lrm, lr, lml, lmh, lcl, lch, lfm, lok, ltk, eqs
INVARIANT
  lim, lrm, lr ∈ BOOL × BOOL × NAT    ∧
  lml, lmh, lcl, lch ∈ NAT × NAT × NAT × NAT    ∧
  lim, lrm, lr ∈ BOOL × BOOL × BOOL × BOOL
OPERATIONS
  ...
  equipment_test ≙
    BEGIN
      eqs := bool  (ltk = true ∧ stk = true ∧ wtk = true ∧
                    (lok = true ∨ sok = true))
    END
END
```

9.3.2 The Korso Method

In the KORSO project a method was developed for the development of correct software [PW95]. The method is designed to cover all phases of the software development, starting with requirements engineering to the development and maintenance of software. Correctness is a relation between two software units. Three types of units are distinguished: Language units, property units and justification units. Property units contain additional information (e. g. algebraic properties) that cannot be expressed within the chosen language. Justification units provide evidence for the correctness of single units or a relation between units.

Software development is represented in a development graph. The nodes of the development graph are units, relations between units are edges in the development graph. The development proceeds in development steps that change the development graph. Figure 9.2 shows an example. On the left-hand side, a unit `A` uses another unit `B` connected by a used-by relation σ. The unit `B` has been developed by an development step ϱ to `B'`. The unit `A` shall use this newly developed unit `B'` instead of `B`. On the right-hand side we see the result: unit `A` is developed by a derived development step ϱ' to a unit `A'`, which now is related to `B'` by a used-by relation σ'. Note that the development step ϱ' has a *justification* J attached. Further development steps and example developments can be found in [PBDD95] and [BDDG93].

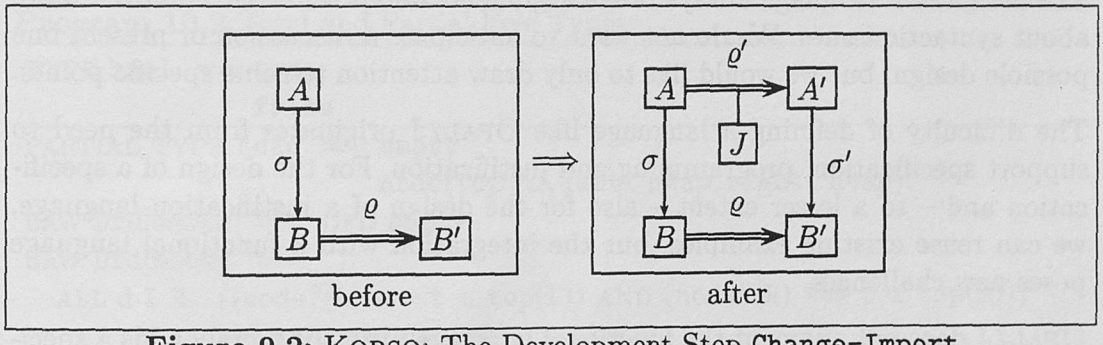

Figure 9.2: KORSO: The Development Step `Change-Import`

The KORSO method is interesting for two reasons. One is the introduction of justification units. This indicates that justifications are regarded as an integral part of a software project (a "first class citizen"), the same as specification and program. Justification units can be developed in the same way (of course by different steps) as the other units. The other characteristic feature is the inclusion of formal, semi-formal and even informal documents in the development process, which is complemented by the introduction of formal and pre-formal development steps.

Chapter 10

Further Topics

In this thesis we concentrated on the architecture of a compiler with integrated support for justifications. So we had to leave out the treatment of some interesting topics, which are presented briefly in this chapter.

10.1 Language Design

The design of a language always raises arguments about semantic and even more about syntactic issues. We do not want to anticipate a discussion or present one possible design, but we would like to only draw attention to some specific points.

The difficulty of defining a language like OPAL/J originates from the need to support specification, programming and justification. For the design of a specification and – to a lesser extent – also for the design of a justification language, we can reuse existing examples, but the integration with a functional language poses new challenges.

OPAL/J does not augment the functional language OPAL, it merely adds a specification and a justification language and defines the specification and justification semantics for the existing language. So we have to define the specification part of the semantics for the "worst-case" situation. Unfortunately this leads sometimes to awkward situations.

We describe one problem in more detail, namely the declaration of free types. Some other areas are only briefly discussed.

Declaration of Free Types To illustrate this example, we compare the free types of heaps and Boolean values. Free-type constructors are *not* necessarily total functions. The free-type constructor for heaps is indeed partial: the top

value and the left and right sub heap must together form a valid heap again[1]. Hence, we cannot generate definedness axioms for free types automatically. The appropriate definedness axioms have to be added explicitly by the user. This is to be expected for the heap type – where it is fine – but we cannot generate axioms DFD true or DFD false for bool either, which is very annoying.

Program 10.1 Definedness Axioms and Free Types

```
TYPE bool == true
             false
LAW Dfd[true] == DFD true
LAW Dfd[false] == DFD false
```

```
TYPE heap == empty
             node(top: α, left: heap, right: heap)
LAW Dfd[empty] == DFD empty
LAW Dfd[node] ==
  ALL d L R. ((node?(L) ⟹ t < top(L)) AND (node?(R) ⟹ t < top(R)))
             ⟹ DFD node(t, L, R)
```

Since most free types do have total constructors, the obligation to add these definedness laws explicitly can be become very tiresome. So it would be nice to have total constructors as the default, with a new keyword to designate partial constructors (with explicit definedness conditions).

Program 10.2 Total and Partial Free Types

```
TYPE bool == true
             false
```

```
PARTIAL TYPE heap == empty
                     node(top: α, left: heap, right: heap)
LAW Dfd[empty] == DFD empty
LAW Dfd[node] ==
  ALL d L R. ((node?(L) ⟹ t < top(L)) AND (node?(R) ⟹ t < top(R)))
             ⟹ DFD node(t, L, R)
```

Declaration of Functions Modern specification languages like Z or B introduce a lot of notations for the declaration of functions (different arrows) that allow to specify at the same time other properties of the declared function, such as partiality, surjectivity, etc. In functional programming (or other specification languages) these distinctions are not made. Proving might be easier, though, if a function is known to be total, so it might be worth to introduce additional syntax.

[1] Obviously, the constructor is of limited use for this type. But the free type declaration allows the use of selectors, discriminators and pattern matching.

Axiom vs. Theorem In specifications, the role of formulas for proving is not important. Models have to satisfy all these formulas and it does not matter whether some formulas are consequences of others. For proving, the difference between axioms and theorems is important, however. Axioms must be proven to be non-contradictory, which is a difficult task, and often deferred. If finally a model of the axioms is developed, the proof that the model does fulfil the axioms has as a corollary the consistency of the axioms.

There is some controversy whether formulas should be designated as axioms or as theorems. Often there is more than one possible set of axioms for a given set of formulas, and there is the opinion that the specifier should not restrict the choice for the later verification. However, if the justification is part of the unit, the choice should be recorded in the source code.

Pathological Cases There are some pathological cases that must be handled by the language definition. Empty sorts are a well-known example. If a bound variable ranges over an empty sort, the whole formula is true, which can lead to counterintuitive results. Another pecularity concerns the bottom completion of types to handle undefinedness. Consider the function space $(t_\perp \to t'_\perp)_\perp$ of functions that map values of type $t_\perp$ to values of type $t'_\perp$. There is a subtle difference between the bottom element of the function space $\perp_{t_\perp \to t'_\perp}$ and the everywhere undefined function U defined by $U(x) \longmapsto \perp_{t'}$ for all x.

These pathological cases can be circumvented by additional context conditions. ISABELLE/HOL solves the problem of empty sorts this way: the user is required to provide a witness for every sort. For OPAL/J this is not an option, because empty sorts are not forbidden now in OPAL. If we add such a context condition to OPAL/J, it would no longer be a proper extension of OPAL.

10.2 Justification Support

10.2.1 Proof Obligations for Data-Type Implementation

From the examples presented in Chapter 7 we can conclude that the automatic check of the correctness of data-type implementations is an important advantage of a correctness-aware compiler. Unfortunately, there are a lot of proof obligations generated to check the congruence and the closedness properties. The compiler should be able to recognize important special cases, so that only the necessary proof obligations are constructed.

Let T be the type declaration from the interface and T' the type implementation. Then some special cases are:

- T and T' are identical. No proof obligations are necessary.
- T is a free type. The visibility predicate can be generated by the compiler.
- There is an isomorphism $i : T \leftrightarrow T'$. The user must prove the isomorphism property, but no other proof obligations are necessary.
- T is a free type and T' is an extension of T by additional constructors. The congruence properties need not be proven.

It remains to be seen which special cases are important for practical software development.

10.2.2 Test Heuristics for Functional Programs

Testing imperative programs is well-studied, but the notions, algorithms and heuristics do not always carry over to the area of functional programming. Of course, black-box testing is independent of the implementation and the programming language used and techniques developed for black-box testing are directly applicable. However, black-box testing should be complemented by implementation-dependent white-box testing. White-box testing functional programs requires a re-examination of the notions developed for testing imperative programs.

Higher-order functions pose another challenge to the testing of functional programs. First, it is not clear how the heuristics used for flat data types carry over to function spaces; so the definition of test cases is difficult. Second, the generation of functions as test data for higher-order arguments is non-trivial. Finally, the check whether functions fulfill certain properties, e. g. during test evaluation or an automatic check of test data with respect to the test-case predicates, is in general undecidable.

Parameterization and polymorphism introduce an additional problem. Polymorphic functions are not executable and hence, not testable. Before testing polymorphic functions, one has to choose an instance, but this instance has to be chosen with care. We have seen in Section 9.1.2 that the validity of a formula may depend on the instance chosen. So one instance will in general not suffice. For parameterization the same considerations hold, with the added complexity that also functions may be parameters.

10.2.3 Tacticals for Batch Proofs

The tacticals found in existing theorem-provers are often not suited for the construction of batch proofs. Information about the origin of formulas is not pre-

served. This is not important for interactive proofs, where the user knows about the significance of the various formulas. But batch proofs and interactive proofs have different requirements.

For illustration, consider the nested induction proof for the commutativity of +: `LAW commu == ALL x y. x + y ≡ y + x`. The basic tactical allows only for induction on a single variable, which results in the following proof state:

```
targets: <§0 (ALL y. 0 + y === y + 0) AND
(ALL pred_4_c. ALL y. succ(pred_4_c + y) === y + succ(pred_4_c))>
premises: <§0 ALL Y_6. 0 + Y_6 === Y_6,
§1 ALL pred_4_a0 Y_6. succ(pred_4_a0) + Y_6 === succ(pred_4_a0 + Y_6)>
```

The premises are the definitional equation of +, the targets are the result of induction after the first variable x. For the nested induction, we must apply the induction tactic for both occurrences of y, but not for the variable `pred_4_c`. This is easily achieved in an interactive proof, because the user has the knowledge which variable the induction should be carried out. But in a batch proof all variables are treated equally, and it is not possible to restrict the application of the induction rule to the proper places.

In the tactics library for OPAL/J, we introduced a tactic (`mInduct`) that applies simultaneous induction to all variables at once. The resulting formula is long and very complicated and not suited for interactive proofs, but in batch proofs this is no obstacle.

10.2.4 Debugging Proof States

The example proof states shown in Chapters 6 and 7 do exhibit the reasons for the mistakes made, but on the other hand one recognizes that a lot of skill is needed in the interpretation of the formulas. Consider the proof state shown in Figure 6.14 on Page 99. It is important not to be intimidated by the number of formulas, and to be able to classify formulas: some are given explicitly, some result from definitional equations, and others represent the induction hypotheses and induction targets.

Some requirements for a user interface of a debugger turned up during the development of the example justifications.

- Navigation through the formal proof is a basic requirement for debugging proof states.
- We need to have two representations for a formula: First, the low-level representation, i. e. the exact structure of the formula, but second, also a

higher-level description is needed that describes the "meaning" of a formula, e. g. "induction base case #2".

- Another important piece of information is the "history" of a formula, i. e. the sequence of tactics that introduced the formula into the proof state.
- Some kind of selective display of sequents is needed, because the textual representation of a full proof state is very long[2].
- Highlighting the redex of a tactical and the changed formulas in the following proof state helps understanding how the theorem-prover works.

OPAL/J uses the EMACS editor with a dedicated mode and achieves some of the above goals. The mode differentiates with different colours between formulas that were part of the goal (like LAW `ascending_◇` and the formulas that are derived from the free type property), and those formulas that were introduced during the proof. OPAL/J supports higher-level representations at least for the initial formulas of a proof.

A more sophisticated graphical user interface has additional possibilities. We might change the representation of a formula on a mouse click, and might also get more information about the history of a formula. Very important is the possibility of a graphical representation of a proof; the user interface of KIV can serve as an example. The textual representation of a formal proof is necessarily linear, a graphical user interface could reconstruct the tree structure of a proof and thus facilitate the understanding of the proof.

10.3 Miscellaneous

10.3.1 Semi-Formal Specifications

The justification methods presented in Section 6 consist of formal, semi-formal and informal methods. This classification carries over to specifications. Most often specification implicitly means *formal* specification, but of course semi-formal and informal specifications have a place in software development, too.

The support of semi- and informal specifications is difficult for several reasons. These kinds of specifications are often centred around graphical notations, whereas formal specifications and programs are linear text. The handling of graphical notations is not standard in usual compilers, the combination of linear text and graphical notations requires different editors.

[2]The full representation of a formal proof can easily exceed several thousand lines.

Another difficulty is the integration of semi-formal and informal specification techniques into the semantics of the programming language[3] and to handle the development of informal and semi-formal software units (see also [Hußß95, Hußß97]). Typical issues concern the development of semi-formal specifications, the implementation of programs with respect to a semi-formal specification, and, most important, the justification that an implementation conforms to a semi-formal specification.

10.3.2 Optimization

Up to now our only concern was ensuring the correctness of the final software product, but "Specifications Can Make Programs Run Faster" [Van93], i. e. specifications can also be useful to enhance the *efficiency* of the generated programs. In functional programming, the most important optimizations concern transformation of recursion to tail recursion that can be translated to iterative loops in the compilation process. [BW82] and [Par90] contain transformations like the one in Figure 10.1, which have algebraic application conditions. If these application conditions are available – e. g. because the programmer included appropriate justifications – the compiler is able to generate more efficient code.

DEF $f(x)$ == IF $b(x)$ THEN $h(x)$ ELSE $k(x) \circ f(k'(x))$ FI
Application condition: $\circ$ is associative and has a neutral element 0
DEF $f(x)$ == $f'(0, x)$ DEF $f'(y, x)$ == IF $b(x)$ THEN $y \circ H(x)$ ELSE $f'(y \circ k(x), k'(x))$ FI

Figure 10.1: The Simplification Rule of Linear Recursion

These considerations also hold for imperative programs. [Van93] describes a prototype compiler for a small imperative language that uses specifications to help in side effect analysis, constant subexpression elimination and other optimizations.

[3] We do *not* want to give a formal semantics to graphical specifications (though that would be an interesting topic, too), because that would not be a semi-formal specification anymore.

Chapter 11

Conclusion

We have presented an approach for the integration of correctness checks into the compilation process. The integration of correctness checks encourages a development style that views correctness checks as a natural part of software development. If correctness checks are performed by separate tools, there is always the danger that these tools are not fully compatible to the compiler, and the temptation to omit the correctness check because the software product must be finished fast is often irresistible.

We summarize the most important features:

- We support *different methods of justification*, most notably formal testing and formal proofs. Often only a single method is used during the development of correct software. In our view, different justification methods are not mutually exclusive but rather complementary. The support for different justification methods allows the user to choose the appropriate justification method for different situations.

- Justification support is *integrated into the compilation process.* In the development of compilers, more and more correctness checks have been automated and integrated into the compiler, from the check for unreachable code to parameter type mismatches. We see the integration of justification support as a natural extension of a compiler's tasks.

- Our approach advocates *literate justification*, i. e. the justification is embedded in the source code. Related information is kept together. The development of specification, implementation and justification can be performed in parallel. Since the user develops the justification, the compiler need only check the justification. This makes the approach feasible; an automated computation of justifications is too expensive.

- The semantics of a unit is an algebraic specification $\mathcal{S} = (S, OP, F)$. *A programming unit is correct, iff its semantics is a consistent algebraic specification.* Correctness can be proven by constructing a model or by an algebraic relation. The first method is chosen for units that are translated to object code, the latter is used to prove correctness for interface units.

- The most important relation is the *implementation relation.* The algebraic definition suggests that formulas from the implementation are inserted as theorems into the interface. We propose instead to add the *lifted formulas* of the interface to the implementation.

- The user declares for each proof obligation separately the axioms that are used to prove it. These *proof declarations* are part of the source code.

- Units are developed independently. Our notion of *relative correctness* translates this principle to the semantic level.

- The justification component is an analysis phase and should be inserted between the context checker and the optimizer. *Phases of the justification component are: collection of formulas, unit correctness check, and justification correctness check.*

- The *implementation of the testing component* requires the *integration of an interpreter.* The interpreter enables the compiler to perform the test execution before the object code has been generated.

- The *implementation of the formal-proof component* can use a third-party theorem-prover or a specialized theorem-prover. We suggest to *implement the theorem-prover as a library in the compiled language.* This allows the implementation of a theorem-prover that can be controlled by the user in the same language that is used for the implementation. Our experience shows that the cost for the implementation is similar to that for the adaptation of an existing theorem-prover.

The approach does not depend on the compiled language being a *functional* programming language, but it does favour the functional paradigm. The algebraic semantics is close to the programming language, literate justification is easier to integrate and security concerns raised by the integration of an interpreter especially for test execution are not so serious for a (pure) functional programming language. However, these considerations do not prohibit the transfer of our approach to other languages. JAVA, with its security model, might be a suitable candidate.

A compiler with integrated justification support cannot replace thorough treatment of verification and validation throughout the software development process. Still, the integrated justification support aids software engineers. Experi-

ence shows that *support for justification checks helps us to avoid some nasty and cryptic errors.* Two of the examples in Chapter 1 were inspired by real-life bugs that it took several days to track down.

A compiler that checks algebraic context conditions extends the options for language developers and software programmers. Currently, algebraic specifications are rarely used. OPAL's support for data-type implementation with junk elements and multiple representations is still an exception. Because the violation of the application conditions is not checked but can lead to cryptic errors, algebraic context conditions must be used with caution. A compiler with integrated support for justification checks makes the use of algebraic conditions more agreeable, because errors are checked at compile time and we can benefit from the enhanced possibilities.

Bibliography

[Abr96] Jean-Raymond Abrial. *The B-Book.* Cambridge University Press, 1996.

[AH77] Kenneth Appel and Wolfgang Haken. The Solution of the Four-Color-Map Problem. *Scientific American*, 237(4):108–121, October 1977.

[B95] B-Core (UK). The B-Toolkit Demonstration. In Peter D. Mosses, Morgan Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT '95 Proceedings*, LNCS, pages 805–806. Springer, 1995.

[BBC+00a] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giméneza, Hugo Herbelin, Gérard Huet, Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant, Reference Manual, Version 6.3.11. Technical report, INRIA Rocqencourt, May 2000.

[BBC+00b] Nikolaj Bjorner, Anca Browne, Michael Colon, Bernd Finkbeiner, Zohar Manna, Henny Sipma, and Tomas Uribe. Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. Formal Methods in System Design, 2000.

[BDDG93] Ralph Betschko, Sabine Dick, Klaus Didrich, and Wolfgang Grieskamp. Formal Development of an Efficient Implementation of a Lexical Scanner within the KorSo Methodology Framework. Technical Report 93–30, TU Berlin, October 1993.

[BG99] Robert M. Balzer and Neil M. Goldman. Mediating Connectors: A Non-By Passable Process Wrapping Technology. In *Information Survivability 99*, 1999.

[BH96] Michel Bidoit and Rolf Hennicker. Behavioural Theories and the Proof of Behavioural Properties. *Theoretical Computer Science*, 165(1):3–55, 1996.

[BH98] Michel Bidoit and Rolf Hennicker. Modular Correctness Proofs of Behavioural Implementations. *Acta Informatica*, 1998.

[BW82] F.L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer Verlag, Berlin, Heidelberg, New York, Berlin, 1982.

[CEW93] Ingo Claßen, Hartmut Ehrig, and Dietmar Wolz. *Algebraic Specification Techniques and Tools For Software Devleopment – The ACT Approach.* AMAST Series in Computing. World Scientific, 1993.

[Cla91] Ingo Claßen. Revision of ACT ONE in LOTOS. Lotosphere Report Lo/WP1/T1.4/TUB/N0001/V1, 1991.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Manadayarn Srivas. *A Tutorial Introduction to PVS.* Computer Science Laboratory, SRI International, 1995. Presented at WIFT '95 (Workshop on Industrial-Strength Formal Specification Techniques).

[DF94] Klaus Didrich and Andreas Fett. Verifying OPAL — Part I: Verifying in the Small. Technical Report 94-23, TU Berlin, December 1994.

[DFG+94] Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. In Jürg Gutknecht, editor, *Programming Languages and System Architectures*, LNCS 782, pages 228–244. Springer, March 1994.

[Did92] Klaus Didrich. *ACT ONE–C*: Eine Erweiterung der Spezifikationssprache ACT ONE im Hinblick auf kompakte Darstellungen von Spezifikationen. Diplomarbeit, TU Berlin, 1992.

[Did99] Klaus Didrich. Compiler Support for Specification and Justification – Description of a Case Study. Technical Report 99-18, TU Berlin, November 1999.

[DK96] Klaus Didrich and Torsten Klein. A Pragmatic Approach to Software Documentation. Technical Report 96-4, TU Berlin, June 1996.

[EEC85] EEC. Council Directive 85/374/EEC of 25 July 1985 on the approximation of the laws, regulations and administrative provisions of the Member States concerning liability for defective products. Official Journal L 210, 1985. pages 0029–0033.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[EM90] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1990.

[FSF00] The Free Software Foundation. The GNU Privacy Guard. `http://www.gnupg.org/`, 2000.

[Gau95] Marie-Claude Gaudel. Testing can be formal, too. In *Proceedings of the 6th International Joint Conference CAAP/FASE (TAPSOFT '95)*, LNCS 915, pages 82–96, 1995.

[GG75] J. B. Goodenough and S. Gerhart. Towards a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, 1(2), June 1975.

[Gib94] Jeremy Gibbons. An Introduction to the Bird-Meertens Formalism. In *New Zealand Formal Program Development Colloquium Seminar*, Hamilton, 1994.

[Gri81] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[Gri95] Klaus Grimm. *Systematisches Testen von Software – Eine neue Methode und eine effektive Teststrategie*. Number 251 in GMD-Berichte. R. Oldenbourg, 1995.

[Gri96] E. Pascal Gribomont. Preprocessing for Invariant Validation. In Martin Wirsing and Maurice Nivat, editors, *Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology, AMAST'96*, LNCS 1101, pages 256–270. Springer, 1996.

[Gün93] Andreas Günther. Produkthaftung für Software – Ein obiter dictum aus den USA. *Computer und Recht*, pages 544–546, 9 1993.

[HNS97] Steffen Helke, Thomas Neustupny, and Thomas Santen. Automating Test Case Generation from Z Specifications with Isabelle. In *Proceedings of the 10th International Conference of Z Users (ZUM'97)*, 1997.

[How87] William E. Howden. *Functional Program Testing & Analysis*. McGrawHill, 1987.

[Hud98] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

[Hue95] Gérard Huet. Type Theory, Specification Languages and Program Verification. International Summer School on Logic of Computation, Marktoberdorf, 1995.

[Hußm95] Heinrich Hußmann. *Formal Foundations for SSADM.* Habilitation, TU München, März 1995.

[Hußm97] Heinrich Hußmann. *Formal Foundations for Software Engineering Methods.* LNCS 1322. Springer, 1997.

[Jon00] Mark P. Jones. Integrating Programming, Properties and Validation. In Roland Backhouse and José Nuno Oliveira, editors, *Mathematics of Progam Construction, 5th International Conference, MPC 2000*, LNCS 1837, page 1 [sic!], 2000.

[JSB+95] Richard Jüllig, Y. V. Srinivas, Lee Blaine, Li-Mei Gilham, Allen Goldberg, Cordell Green, Jim McDonald, and Richard Waldunger. *Specware Language Manual*, 1995.

[Knu83] Donald E. Knuth. The Web System of Structured Documentation. Stanford Computer Science Report CS980, Stanford University, Stanford, CA, 1983.

[KS91] Frank A. Koch and Peter Schnupp. *Software-Recht*, volume 1. Springer, 1991. Chapter 3.6.

[KS98] Stefan Kahrs and Donald Sannella. Reflections on the Design of a Specification Language. In *Proceedings of the International Colloquium on Fundamental Approaches to Software Engineering, ETAPS'98*, LNCS 1382. Springer, 1998.

[KST94] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The Definition of Extended ML. Technical Report ECS-LFCS-94-300, University of Edinburgh, LFCS, August 1994.

[KST97] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The Definition of Extended ML: a Gentle Introduction. *Theoretical Computer Science*, 173:445–484, 1997.

[LGH+78] Ralph L. London, John V. Guttag, James J. Horning, Butler W. Lampson, J. G. Mitchell, and Gerald J. Popek. Proof Rules for the Programming Language Euclid. *Acta Informatica*, 10:1–26, 1978.

[LHL+77] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. L. Popek. Report On The Programming Language Euclid. *Sigplan Notices*, 12(2), February 1977.

[LPPU94] Michael Lowry, Andrew Philpot, Thomas Pressburger, and Ian Underwood. Amphion: Automatic Programming for Scientific Subroutine Libraries. In *Proceedings of the 8th International Symposium on Methodologies for Intelligent Systems*, LNCS 869, pages 326–335. Springer, 1994.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[MBB+99] Zohar Manna, Nikolaj Bjorner, Anca Browne, Michael Colon, Bernd Finkbeiner, Mark Pichora, Henny B. Sipma, and Tomas Uribe. *Tool Support for System Specification, Development and Verification*, chapter An Update on STeP: Deductive-Algorithmic Verification of Reactive Systems, pages 174–188. Advances in Computing Science. Springer, 1999.

[ML84] Per Martin-Löf. *Intuitonistic Type Theory.* Bibliopolis, 1984.

[MLP79] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social Processes and Proofs of Theorems and Programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[Nec97] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages (POPL '97)*, 1997.

[Nic95] Jan Nicklisch. Higher Order Partial Algebras in View of the Semantics of Functional Programs. Master's thesis, TU Berlin, July 1995.

[Nip98] Tobias Nipkow. *Isabelle/HOL – The Tutorial.* TU München, 1998.

[NL98a] George C. Necula and Peter Lee. Efficient Representation and Validation of Proofs. In *IEEE Symposium on Logic in Computer Science (LICS '98)*, 1998.

[NL98b] George C. Necula and Peter Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, 1998.

[Pag88] Clive G. Page. *Professional Programmer's Guide to Fortran77.* Pitman, 1988.

[Par90] Helmut Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development.* Springer Verlag, Berlin, Heidelberg, New York, Berlin, 1990.

[Par95a] Catherine Parent. *Synthèse de preuves de programmes dans le calcul des constructions inductives.* PhD thesis, École normale supérieure de Lyon, 1995.

[Par95b] Catherine Parent. Synthesizing proofs from programs in the Calculus of Inductive Constructions. In *Mathematics for Program Construction '95*, LNCS 947. Springer, 1995.

[Pau87] Lawrence C. Paulson. *Logic and Computation.* Cambridge University Press, 1987.

[PBDD95] Peter Pepper, Ralph Betschko, Sabine Dick, and Klaus Didrich. Realizing Sets by Hash Tables. In Manfred Broy and Stefan Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, LNCS 1009, pages 58–73. Springer, 1995.

[Pep91] Peter Pepper. Transforming Algebraic Specifications – Lessons Learnt from an Example. In B. Möller, editor, *Proceedings of the IFIP TC 2 Conference on Constructing Programs from Specifications*, pages 1–27. Elsevier, 1991.

[PH99] Simon Peyton Jones and John Hughes (editors). Report on the Programming Language Haskell 98. Technical report, Yale University, 1999.

[PHL+77] G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London. Notes on the Design of Euclid. *ACM SIGPLAN Notices*, 12(3):11–18, March 1977.

[Pop72] Karl R. Popper. *The Logic of Scientific Discovery.* Hutchinson of London, seventh impression edition, 1972.

[Pus94] Cornelia Pusch. Verifikation einer Entwicklung von AVL-Bäumen in Isabelle. Diplomarbeit, TU München, 1994.

[PW95] Peter Pepper and Martin Wirsing. A Method for the Development of Correct Software. In Manfred Broy and Stefan Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, LNCS 1009, pages 27–57. Springer, 1995.

[Rei95] Wolfgang Reif. The KIV-Approach to Software Verification. In Manfred Broy and Stefan Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, LNCS 1009, pages 338–368. Springer, 1995.

[Riv90] Ronald L. Rivest. *Handbook of Theoretical Computer Science*, volume A, chapter 13: Cryptography, pages 717–756. Elsevier, 1990.

[RS97] Wolfgang Reif and Gerhard Schellhorn. Theorem Proving in Large Theories. In *Automated Theorem Proving in Software Engineering (CADE-14 workshop)*, 1997.

[RSS97] Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel. Proving System Correctness with KIV 3.0. In William McCune, editor, *Automated Deduction – CADE-14*, LNAI 1249, pages 69–72. Springer, 1997.

[SC96] Phil Stocks and David Carrington. A Framework for Specification-Based Testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[Sha97] Stuart Shapiro. Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering. *IEEE Annals of the History of Computing*, 19(1):20–54, 1997.

[Som92] Ian Sommerville. *Software Engineering.* Addison-Wesley, 4th edition, 1992.

[SP93] Douglas R. Smith and Eduardo A. Parra. Transformational Approach to Transportation Scheduling. In *Proceedings of the 8th Knowledge-based Software Engineering Conference*, pages 60–68. IEEE Computer Society Press, 1993.

[ST89] Donald Sannella and Andrzej Tarlecki. Toward Formal Development of ML Programs: Foundations and Methodology. Technical Report ECS-LFCS-89-71, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.

[Str87] Bjarne Stroustrup. *The C++ Programming Language.* ANSI Based Document; AT&T Bell Telephone Laboratories, Murray Hill. Addison Wesley, Mass., 1987.

[SW98] Ute Schmid and Fritz Wysotzki. Induction of Recursive Program Schemes. In *Proceedings of the 10th European Conference on Machine Learning ECML-98*, LNAI 1398, pages 214–225. Springer, 1998.

[Tho98] Robin Thomas. An Update on the Four-Color Theorem. *Notices of the AMS*, 45(7):848–859, August 1998.

[Van93] Mark T. Vandevoorde. Specifications Can Make Programs Run Faster. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development, Proceedings*, LNCS 668, pages 215–229. Springer, April 1993.

[Van96] Glenn Vanderburg. *Tricks of the Java Programming Gurus*, chapter 23: Pushing the Limits of Java Security. SAMS.net Publishing, 1996.

[vL90] Jan van Leeuwen. *Handbook of Theoretical Computer Science*, volume A, chapter 10: Graph Algorithms, pages 525–632. Elsevier, 1990.

[VW90] Christopher J. Van Wyk. Literate Programming – an Assessment. *Communications of the ACM*, 33(3):361–365, March 1990.

[WB89] Philip Wadler and Stephen Blott. How to make ad–hoc polymorphism less ad hoc. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[WDC+95] Uwe Wolter, Klaus Didrich, Felix Cornelius, Markus Klar, Roland Wessäly, and Hartmut Ehrig. How to Cope with the Spectrum of SPECTRUM. In Manfred Broy and Stefan Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, LNCS 1009, pages 173–189. Springer, 1995.

[Wei97] Thiemo Weiß. Testfallgenerierung auf der Basis von Z-Spezifikationen mit Hilfe der Klassifikationsbaum-Methode. Diplomarbeit, TU Berlin, April 1997.

[Wir90] Martin Wirsing. *Handbook of Theoretical Computer Science*, volume B, chapter 13: Algebraic Specification, pages 675–788. North-Holland, 1990.